

AD-A230 298

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
JAN 03 1991
S D

THESIS

RECOGNITION OF VLSI MODULE
ISOMORPHISM

by

Emmanouil N. Zagourakis

March 1990

Thesis Advisor:

Chayn Yang

Approved for public release; distribution is unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No 0704-0188	
1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited		
2b DECLASSIFICATION / DOWNGRADING SCHEDULE			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
4 PERFORMING ORGANIZATION REPORT NUMBER(S)			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b OFFICE SYMBOL (If applicable) EC	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000			7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		
8a NAME OF FUNDING / SPONSORING ORGANIZATION		8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c ADDRESS (City, State, and ZIP Code)			10 SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO	PROJECT NO	TASK NO
					WORK UNIT ACCESSION NO
11 TITLE (Include Security Classification) RECOGNITION OF VLSI MODULE ISOMORPHISM					
12 PERSONAL AUTHOR(S) ZAGOURAKIS, Emmanouil N.					
13a TYPE OF REPORT Master's Thesis		13b TIME COVERED FROM _____ TO _____		14 DATE OF REPORT (Year, Month, Day) 1990 March	
15 PAGE COUNT 151					
16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	VLSI circuit verification; gate recognition; isomorphism identification		
19 ABSTRACT (Continue on reverse if necessary and identify by block number) The purpose of this study is to determine whether or not a program could be developed to examine isomorphism between parts of a VLSI layout. Many simulation files, obtained through Magic's hierarchical extractor, were analyzed in order to develop a C program to accomplish recognition in several types of gates. This recognition gives signatures in order to check for isomorphism. The development and design of the algorithms used in different parts of the program are described. Results demonstrate that recognition of elements in a CMOS circuit is possible, even with moderate complexity structures. An appendix with the C program listings is included.					
20 DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a NAME OF RESPONSIBLE INDIVIDUAL YANG, Chyan			22b TELEPHONE (Include Area Code) 408-646-2266		22c OFFICE SYMBOL EC/Ya

DD Form 1473, JUN 86

Previous editions are obsolete

S/N 0102-LF-014-6603

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Approved for public release; distribution is unlimited

Recognition of VLSI Module Isomorphism

by

Emmanouil N. Zagourakis
Lieutenant, Hellenic Navy.
B.S., Hellenic Naval Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

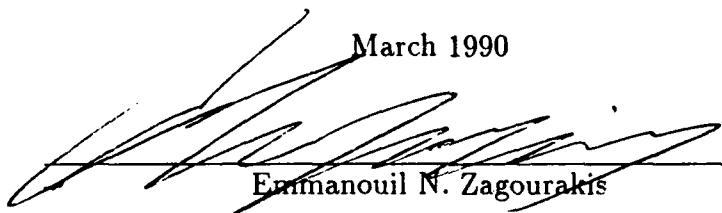
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the


NAVAL POSTGRADUATE SCHOOL


March 1990


Author:


Emmanouil N. Zagourakis

Approved by:


Chyan Yang, Thesis Advisor


Mitchell L. Cotton, Second Reader


John P. Powers, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The purpose of this study was to determine whether or not a program can be developed to examine isomorphism between parts of a VLSI layout. Many simulation files, obtained through Magic's hierarchical extractor, were analyzed in order to develop a C program to accomplish recognition of several types of gates. This recognition gives signatures in order to check for isomorphism.

The development and design of the algorithms used in different parts of the program are described. Results demonstrate that recognition of elements in a CMOS circuit is possible, even with moderate complexity structures. An appendix with the C program listings is included.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification:	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	



TABLE OF CONTENTS

I.	INTRODUCTION	1
	A. BACKGROUND	1
	B. SCOPE OF THE THESIS INVESTIGATION	2
	C. THESIS OUTLINE	3
II.	GRAPH THEORY - THE ISOMORPHISM PROPERTY.	4
III.	ANALYSIS OF SIMULATION FILES.	7
	A. FROM MAGIC TO SIMULATION FILES	7
	B. SIMULATION FILES	8
IV.	ANALYSIS AT TRANSISTOR LEVEL.	11
	A. GROUPING OF TRANSISTORS	11
	B. IMPLEMENTATION	11
V.	EXAMINING .SIM FILES OF KNOWN CMOS CIRCUITS.	14
	A. RECOGNITION ALGORITHMS	15
	1. INVERTER	15
	2. PASSGATE	15
	3. 2-INPUT NOR	16
	4. 2-INPUT NAND	17
	5. 2-PHASE CLOCKED INVERTER	17
	6. PRECHARGED INVERTER	18
	7. 2-INPUT PRECHARGED NOR	19
	8. 2-INPUT PRECHARGED NAND	20
	B. LENGTH AND WIDTH SIGNATURES	21
	C. IMPLEMENTATION	22

D.	PROGRAM ACCOMPLISHMENTS	23
VI.	EXAMINATION OF .SIM FILES FOR UNKNOWN CMOS CIRCUITS. .	24
A.	STRUCTURES WITH PATH TO VDD AND GND	24
B.	STRUCTURES WITHOUT PATHS TO VDD AND GND	25
C.	IMPLEMENTATION	25
VII.	IDENTIFICATION FOR ISOMORPHISM.	28
A.	TRANSISTOR LEVEL	28
B.	GATE LEVEL	29
C.	STRUCTURE LEVEL	30
VIII.	CONCLUSIONS.	31
A.	ISOMORPHISM	31
B.	FUTURE RESEARCH	31
	APPENDIX A: PROGRAM LISTINGS	32
A.	GLOBAL VARIABLES	32
B.	GATE RECOGNITION PROCESS	51
C.	STRUCTURE RECOGNITION	89
D.	ISOMORPHISM VERIFICATION	116
E.	OTHER FUNCTIONS	124
	REFERENCES	141
	INITIAL DISTRIBUTION LIST	142

LIST OF FIGURES

2.1	Example of three graphs which are mutually isomorphic.	5
3.1	A .sim file for an inverter.	8
3.2	A .sim file for an inverter, without labels.	10

ACKNOWLEDGMENT

I would like to express my appreciation to the faculty and staff of the Electrical and Computer Engineering department for providing me with the necessary tools for my future job. I would like to express special thanks to Professor Chyan Yang for his valuable assistance and for providing the necessary guidance and direction in accomplishing this research and formulating this document. I also wish to thank Professor Cotton for his valued assistance as my second reader. Finally, I am most grateful to my wife Poly for her understanding, patience, and support during my studies; and to my daughter Helen for just being mine.

I. INTRODUCTION

A. BACKGROUND

This thesis is an initial step to develop a tool for fast verification of Very Large Scale Integration (VLSI) circuit design and timing verification. The integration of VLSI chips includes many steps, beginning with setting the specifications and continuing to fabrication of the chips. Design verification checks the chip design before fabrication. The verification process is divided into three parts:

- Functional (logic) verification to ensure that the design gives the desired results.
- Physical (layout) verification to ensure that the physical layout obeys the geometric design rules and to validate the proper circuit connectivity.
- Timing verification to validate the path delays and check that they satisfy the duty cycle according to the design specifications.

The design process of a VLSI circuit leads toward the geometric layout. This geometric layout is used to generate the masks from which the chip is fabricated on wafers. The correctness of the geometric layout is crucial and must be checked before the mask generation process. Due to the complexity of the geometric layout mistakes can occur. Circuits that have been visually checked by designers and layout specialists have been found to have missing contacts when entering the mask shop. Errors like this are costly in terms of design time and, therefore, money. Computer aided design (CAD) tools can reduce the time needed for the design process. Development of a fast circuit verifier can reduce the design time.

A circuit verifier ensures that the design is accurate, reducing the chance of faulty chips. There are several studies which attempt to provide design verification in certain stages of the VLSI design process [1][2]. Some of them use the switch-level model and others the device-level model. Although the switch-level model gives faster results, component regularity can give the device level model the advantages of better identification for certain elements. By comparing structures which consist of certain elements, a quick verification can be achieved. For example, if the designer knows that his geometric layout contains 99,336 transistors, 8,952 inverters, 6,747 passgates, 125 XOR gates and so on, a CAD tool can test for these quantities and perform a quick verification. Beside the device counter process, a timing analysis based on gates (or in general, devices) can be faster by using devices than using transistors.

This research is concentrated on an algorithm that will provide designers with circuit verification. This algorithm accomplishes its task by examining the isomorphism property through the graph theory.

B. SCOPE OF THE THESIS INVESTIGATION

The goal of this thesis is to develop an algorithm that examines whether the isomorphism exists between a Complementary Metal Oxide Silicon (CMOS) circuit geometric layout and the expected layout. The expected design description may produce the expected simulation file before the layout process. By this algorithm, errors that have taken place during assembling the whole geometric layout from its distinct parts can be identified.

The algorithm uses a simulation file as input. This simulation file, consisting of transistors, is analyzed. We first perform gate recognition of several types of gates and remove the transistors which belong to these gates from the simulation file. The remaining transistors are grouped as abstract structures (devices). This gate and/or structure

recognition and taxonomy provides classification among transistors. Transistors of the same connection topology are grouped together. The above process is performed in the expected circuit and in the part of the circuit that we want to examine. The algorithm then performs tests for each kind of gate and structure to verify that the two circuits are isomorphic.

C. THESIS OUTLINE

Chapter II introduces graph theory and discusses briefly the isomorphism property.

In Chapter III, the different steps that are required to generate simulation files are examined (Magic-extractor-ext2sim).

The preliminary stage of structuring the transistor level information for later use is presented in Chapter IV.

Examination of simulation files for known CMOS circuits is presented in Chapter V. Several kinds of gates are recognized and their algorithms are analyzed.

Chapter VI provides the algorithm to group the rest of the existing transistors in structures with proper format, for later examination for isomorphism.

Chapter VII discusses the algorithm for the tests that are performed in order to identify if isomorphism exists.

Chapter VIII summarizes the results of this thesis and includes possible uses of this program in circuit and timing verification.

II. GRAPH THEORY - THE ISOMORPHISM PROPERTY.

A graph, G , is a network of nodes or vertices (V) and arcs or edges (E) from some nodes to others or to themselves [3]. According to properties that exist between vertices and edges, graphs are categorized in a taxonomy that is quite lengthy. Some example categories are:

1. Simple, if no self-loops or multiple edges exist.
2. Directed, if the pair of end points of an edge is ordered.
3. Euler, every edge appears once if in an undirected graph.
4. Complete, if every pair of distinct vertices is adjacent.
5. Tree, if the graph has no cycles

and many others [3].

A graph can be used to represent a VLSI circuit layout, with transistors as the vertices of a graph and the connections between the transistors its edges. The only difference that exists is that the graph of a VLSI circuit, containing perhaps 100,000 transistors, cannot be placed in any of the above categories. The connectivity follows functional and topological requirements. These functional requirements establish the way of connectivity between the transistors and thus no assumption can be made in order to fit somewhere in graph theory's taxonomy.

In graph theory two concepts of "sameness" exist. These concepts are equality and isomorphism. "Two graphs, G and G' are equal if they have equal vertex sets and

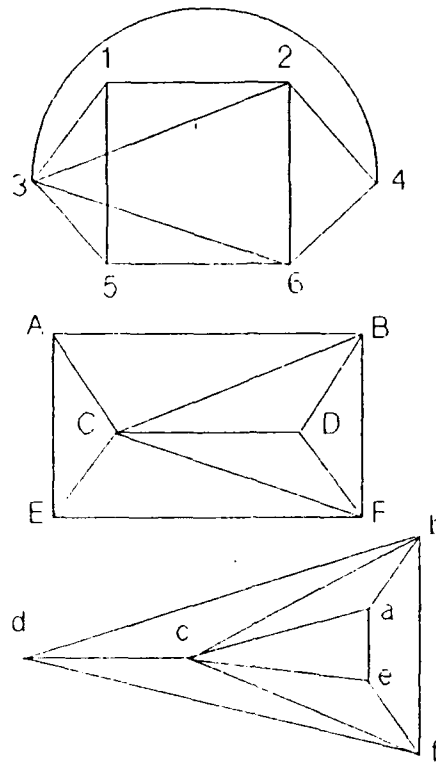


Figure 2.1: Example of three graphs which are mutually isomorphic.

equal edge sets" [4]. The isomorphic property is more fundamental one. "Isomorphic" has Hellenic roots meaning "same structure." Two graphs, G and G' , are said to be isomorphic if a 1-to-1 mapping of graph G' onto graph G exists [5]. The verification of 1-to-1 mapping is not an easy task, especially if the number of vertices and edges is large. The three graphs in Figure 1 are isomorphic, although they don't seem to be at first glance. In fact, the graph isomorphism problem is NP-complete [6].

Isomorphism preserves a number of properties [4], some of which will be useful to this research. These are:

- Same number of vertices.

- Same number of edges.
- Same number of pieces (subgraphs).

If one of the above properties does not exist between two graphs, they are not isomorphic. In other words, either an attempt can be made to check for isomorphism or to try to prove that one of the above properties does not exist and the graphs are not isomorphic. The existence of isomorphism between two graphs, is based on "graph invariants." These are distinct properties or parameters that must exist in both graphs if they are isomorphic. The more graph invariants that can be identified, the more certainty exists about isomorphism. But this is a postulation, not a proof. A "complete graph invariant" is considered the one property that must exist between two isomorphic graphs, and is the only way to prove isomorphism. Since the discovery of such complete graph invariant, in a computable way, would solve the isomorphism problem, the search has been persistent and still continues [7][8].

In this research, the attempt to identify isomorphism is based on assigning graph invariants and examining for existence of the three properties mentioned before. This attempt is made through circuit layout gate recognition. *The taxonomy of different structures of transistors into gates, implies all three properties.* For example, suppose 1,932 inverters exist in a circuit. This means that by recognizing structures as inverters, in the two graphs, and comparing them we have already examined for the first two properties (same vertices and same edges). Furthermore, having the same number of inverters, the third property is also examined. Of course, as mentioned before, this is a postulation not a proof as the gates may be connected in the wrong way. The larger the number of structures which are identified, the less the uncertainty is. This thesis treats the isomorphism at the gate level but more levels can be applied to prove whether isomorphism exists or not.

III. ANALYSIS OF SIMULATION FILES.

The simulation file of a CMOS layout plays a very important role in this thesis. The simulation file is the input to the verification program which was implemented. Since it is used as the source for the needed information for the CMOS layout, it is necessary to discuss the way it is created and the information contained in its lines.

A. FROM MAGIC TO SIMULATION FILES

Computer-aided design (CAD) tools play an important role in the design of VLSI layouts. Several tools have been developed to assist VLSI designers. One such tool for CMOS technology layouts is Magic [9].

Magic was developed by the faculty and students at University of California, Berkeley (UCB) in 1983. It is an interactive system for creating VLSI circuit layouts. It consists of interactive packages, the most important functions of which are:

1. Mead-Conway style of design. This implies simplified design rules and circuit structures.
2. Capability for designs of Manhattan-style layouts. This means that the edges in the layout are vertical or horizontal.
3. Capability for MOSIS-scalable CMOS design rules. Magic's name for this technology is SCMOS.
4. Hierarchical circuit extraction used to convert the graphical layout into a Caltech Intermediate Form (CIF) file.

In order to generate a simulation file, several steps must occur. A VLSI layout can be designed using Magic. The result of this design is a file with the extension .mag.

```

| units: 150 tech: scmos
p D Vdd Q 2 4 78 8
n D GND Q 2 4 78 -10
R GND 839
C Q GND 58
R Q 878
C D GND 13
R D 488
C Vdd GND 47
R Vdd 919

```

Figure 3.1: A .sim file for an inverter.

By using the extractor program which is in the Magic package, we come up with a CIF representation of the file. The extension of this file is now .ext. This .ext file contains informations such as transistor dimensions, circuit connectivity, and resistance and capacitance values between the different nodes of the circuit. The .ext file, which is the hierarchical representation of the original layout, is converted to its flat representation. This is the simulation file and it has extension .sim. This transformation is done by another UCB tool named ext2sim.

This discussion about Magic does not widely cover the subject, and the interested reader may find a lot of detailed information in [10].

B. SIMULATION FILES

It is of great importance to identify the information provided by the simulation file. Figure 3.1 illustrates the .sim file of an inverter.

The .sim file begins with a header line which contains the scale factor (150) and the technology (scmos) of the circuit. Some simulation files may have another field describing the format. In this thesis, scmos technology is required in the technology field (Berkeley or MIT).

Next, the transistor lines follow. Each transistor has its own line, including information about this transistor. Eight fields exist in the transistor line, which, in sequence, are:

1. Transistor type (p or n).
2. Gate field, which contains the node label for the gate.
3. Source field, which contains the node label for the source.
4. Drain field, which contains the node label for the drain.
5. Length field, which contains the scaled length of the transistor.
6. Width field, which contains the scaled width of the transistor.
7. X-location, field which indicates the location of a center point inside the transistor.
8. Y-location, field which indicates the location of a center point inside the transistor.

In this thesis, the first six fields are considered to provide us with useful information. The labels Vdd and GND which appear in a .sim file, indicate a connection with the common voltage source and ground of the circuit. This label is placed either in the source or in the drain field. This placement is performed by the extraction process. It is done in a manner depending on the position of the gates in the layout. For example, a gate can be rotated, inverted and so on. Accordingly, the Vdd or GND label can be interchanged between source and drain fields. That has nothing to do with the functionality of the transistors. A certain gate remains the same no matter where the Vdd and GND are placed.

```
| units: 150 tech: scmos
p 3_147_529 Vdd 3_955_744 2 4 78 8
n 3_147_529 GND 3_955_744 2 4 78 -10
R GND 839
C 3_955_744 GND 58
R 3_955_744 878
C 3_147_529 GND 13
R 3_147_529 488
C Vdd GND 47
R Vdd 919
```

Figure 3.2: A .sim file for an inverter, without labels.

extraction process assigns numerical labels to all of them. An example of such assignment is illustrated in Figure 3.2. Because of this, there are no unlabeled nodes within a layout. In our further research we will consider that labels exist in all fields in a transistor line.

IV. ANALYSIS AT TRANSISTOR LEVEL.

Initial research was conducted to specify the most efficient approach. This was needed not only to achieve the final goal of identification for isomorphism, but also decrease the execution time of the program.

A. GROUPING OF TRANSISTORS

As it was stated in Chapter 2, the more graph invariants (signatures) we identify in a circuit layout, the more certainty exists about the isomorphism property. Initially, the transistors are divided into two categories, P and N transistors. Furthermore, each of these categories can be divided into four categories:

1. P1: P transistors without connection to Vdd.
2. P2: P transistors with connection to Vdd.
3. N1: N transistors without connection to GND.
4. N2: N transistors with connection to GND.

Notice that there are no P transistors connected to GND nor N transistors connected to Vdd due to the complementary technology of CMOS.

B. IMPLEMENTATION

A C program (see Appendix A) was written to implement this preliminary process.

The first step in this program is to accept a .sim file as input. The program start running by the command "iso" followed by one or two file names of .sim files. If one filename is invoked, the program performs just device counting. If two filenames are

invoked, it performs examination for isomorphism between them. These two files have to be in order. That means, the first file cannot be smaller than the second one. From now on we are going to refer to them as big (the first file) and small (the second file) even though they might be equivalent (having same number of transistors).

The program then, opens this file (or these files), and creates a file named "isooutput", which contains all information from the recognition process. The information contains the labels for input, output, phase and whatever else is helpful for the program or the designer. It will also contain all error messages that may be prompted during the program's execution.

After the program accepts its inputs, it first checks that they are in proper format. This is done by examining the headers of the .sim files. In case the specifications are not met, an error message is prompted and the program is terminated.

Then the program creates dummy head and tail pointers for the four linked lists. These four lists are going to contain information for the transistors. Next, the rest of the input files are read. Each line is read and all information stored in a temporary structure. The process then continues the following 8 steps:

1. The first field is examined to see whether it is p or n.
2. If it is p, it is examined for existence of Vdd label in either source or drain fields.
3. If it has a Vdd label, this label is placed only in the source field. Then the remaining of information is placed in a structure which is pointed to by the P2 linked list.
4. If it does not have a Vdd label, the information on this transistor is placed in a structure which is pointed to by the P1 linked list.
5. If the first field is n, it is examined for GND label in either source or drain fields.

6. If it has a GND label, this label is placed only in the source field. Then the rest of the information is placed in a structure which is pointed by the N2 linked list.
7. If it does not have a GND label, the information on this transistor placed in a structure which is pointed to by the N1 linked list.
8. Read in the next line. If transistor information is in the line go to step 1; otherwise the program continues with the recognition process.

V. EXAMINING .SIM FILES OF KNOWN CMOS CIRCUITS.

An important goal of this research was to determine whether or not known CMOS circuits can be recognized. Previously-conducted research [10] showed that this could be done and also can be extended to more types of circuits. This thesis recognizes the following 8 types of gates:

1. Inverter.
2. Passgate.
3. 2-input NOR.
4. 2-input NAND.
5. Precharged inverter.
6. 2-phase clocked inverter.
7. 2-input precharged NOR.
8. 2-input precharged NAND.

The major factor for the selection of the above gates, was the existence of these gates in the VLSI circuits library at NPS. Integrated tools can be used for the benefit of NPS students, if these tools can cooperate and function together.

The circuits of these gates were designed using MAGIC and, through the extraction process, were hierarchically presented. The extracted files were converted to UCB CIF format files, using the ext2sim procedure [9]. The ext2sim program created the

needed .sim files. These simulation files were analyzed to identify the algorithm for their recognition.

Several tests were conducted with .sim files. These files were either selected from the NPS student projects or created for the purpose of this research. Testing files were necessary because, for each type of gate, many arrangements of the labels in the different fields may exist. Because the extraction process places the source and drain field labels interchangeably we must carefully choose various testing files in order to exhaustively test all possible configurations of these gates.

We discuss further details of the recognition process of each gate type in the next two sections.

A. RECOGNITION ALGORITHMS

The recognition algorithms are described next. The algorithms use the four transistor lists which are named P1, P2, N1 and N2. In some gates, there are more than one transistor from the same linked list. This case is resolved by referring to N11, N12 for N1 list, for example. The algorithms are:

1. INVERTER

1. Obtain a p-type and an n-type transistor, from the P2 and N2 linked lists, respectively.
2. If gates and drains of P2 and N2 are the same, an inverter is found; otherwise, continue with a different set of p and n transistors, taken from the P2 and N2 lists.
3. If no other set remains, exit the inverter recognition process.

2. PASSGATE

1. Obtain a p-type and an n-type transistor, from the P1 and N1 linked lists, respectively.

2. If:

- (a) Sources of P1 and N1 are the same, and different from all other fields.
- (b) Drains of P1 and N1 are the same, and different from all other fields.
- (c) Gates of P1 and N1 are different,

then, a passgate is found; Otherwise, continue with a different set of p and n transistors from P1 and N1 lists.

3. 2-INPUT NOR

1. Obtain two p-type and two n-type transistors from P1, P2, N2 and N2 linked lists.

We will refer to each N2 as N21 and N22, respectively.

2. If:

- (a) Gates of P2 and N21 are the same, and different from all other labels.
- (b) Gates of P1 and N22 are the same, and different from all other labels.
- (c) Drains of N21 and N22 are the same, and

- Drain of P22 is the same as the source of P1.
- Drain of N21 (or N22) is the same as the drain of P1,

or,

- Drain of P2 is the same as the drain of P1.
- Drain of N21 (or N22) is the same as the source of P1,

then, a 2-input NOR gate is found; Otherwise, continue with different set of transistors, taken from P1, P2, N2 and N2 lists.

3. If no other set remains, exit the 2-input NOR gate recognition process.

4. 2-INPUT NAND

1. Obtain two p-type and two n-type transistors from P2, P2, N1 and N2 linked lists.

We will refer to each P2 as P21 and P22, respectively.

2. If:

(a) Gates of P21 and N2 are the same, and different from all other labels.

(b) Gates of P22 and N1 are the same, and different from all other labels.

(c) Drain of P21 and P22 are the same, and

- Drain of P21 (or P22) is the same as the source of N1.
- Drain of N2 is the same as the drain of N1,

or,

- Drain of P21 (or P22) is the same the drain of N1.
- Drain of N2 is the same as the source of N1,

then, a 2-input NAND gate is found; Otherwise, continue with different set of transistors, taken from P2, P2, N1 and N2 lists.

3. If no other set remains, exit the 2-input NAND gate recognition process.

5. 2-PHASE CLOCKED INVERTER

1. Obtain two p-type and two n-type transistors from P1, P2, N1 and N2 linked lists, respectively.

2. If:

(a) Gates of P2 and N2 are the same, and different from all other labels.

(b) Gates of P1 and N1 are different from each other and from all other labels,
and

- Drain of P2 is the same as the source of P1.
- Drain of N2 is the same as the source of N1.
- Drain of P1 is the same as the drain of N1,

or,

- Drain of P2 is the same as the source of P1.
- Drain of N2 is the same as the drain of N1.
- Drain of P1 is the same as the source of N1,

or,

- Drain of P2 is the same as the drain of P1.
- Drain of N2 is the same as the source of N1.
- Source of P1 is the same as the drain of N1,

or,

- Drain of P2 is the same as the drain of P1.
- Drain of N2 is the same as the drain of N1.
- Source of P1 is the same as the source of N1,

then, a 2-phase clocked inverter is found; Otherwise, continue with different set of transistors, taken from P1, P2, N1 and N2 lists.

3. If no other set remains, exit the 2-phase clocked inverter recognition process.

6. PRECHARGED INVERTER

1. Obtain a p-type and two n-type transistors from P2, N1 and N2 linked lists, respectively.
2. If:

- (a) Gates of P2 and N2 are the same.
- (b) Gates of P2 and N2 are different from all other labels, and

- Drain of P2 is the same as the source of N1.
- Drain of N2 is the same as the drain of N1,

or,

- Drain of P2 is the same as the drain of N1.
- Drain of N2 is the same as the source of N1,

then, a precharged inverter is found; Otherwise, continue with different set of transistors, taken from P2, N1 and N2 lists.

- 3. If no other set remains, exit the precharged inverter recognition process.

7. 2-INPUT PRECHARGED NOR

- 1. Obtain one p-type and three n-type transistors from P2, N1, N1 and N2 linked lists. We will refer to each N1 as N11 and N12, respectively.

- 2. If,

(a) Gates of P2 and N2 are the same, and different from all other labels.

(b) Gates of N11 and N12 are different from each other, and from all other labels.
and

- Drain of P2 is the same as the drain of N11 and the drain of N12.
- Drain of N2 is the same as the source of N12 and the source of N11.

or,

- Drain of P2 is the same as the source of N11 and the drain of N12.
- Drain of N2 is the same as the source of N12 and the drain of N11.

or,

- Drain of P2 is the same as the source of N11 and the source of N12.
- Drain of N2 is the same as the drain of N11 and the drain of N12,

or,

- Drain of P2 is the same as the drain of N11 and the source of N12.
- Drain of N2 is the same as the drain of N12 and the source of N11,

then, a 2-input precharged NOR gate is found; Otherwise, continue with different set of transistors, taken from P2, N11, N12 and N2 lists.

3. If no other set remains, exit the 2-input precharged NOR recognition process.

8. 2-INPUT PRECHARGED NAND

1. Obtain one p-type and three n-type transistors from P2, N1, N1 and N2 linked lists. We will refer to each N1 as N11 and N12, respectively.

2. If,

(a) Gates of P2 and N2 are the same and different from all other labels.

(b) Gates of N11 and N12 are different from each other and from all other labels,

and

- Drain of P2 is the same as the source of N11.
- Drain of N2 is the same as the source of N12.
- Drain of N11 is the same as the drain of N12, and different from all other labels,

or,

- Drain of P2 is the same as the source of N11.

- Drain of N2 is the same as the drain of N12.
- Drain of N11 is the same as the source of N12, and different from all other labels,

or,

- Drain of P2 is the same as the drain of N11.
- Drain of N2 is the same as the source of N12.
- Source of N11 is the same as the drain of N12, and different from all other labels,

or,

- Drain of P2 is the same as the drain of N11.
- Drain of N2 is the same as the drain of N12.
- Source of N11 is the same as the source of N12, and different from all other labels,

then, a 2-input precharged NAND gate is found; Otherwise, continue with different set of transistors, taken from P2, N11, N12 and N2 lists.

3. If no other set remains, exit the 2-input precharged NAND gate recognition process.

B. LENGTH AND WIDTH SIGNATURES

Through the recognition process of each gate, another process is also performed. This process consists of placing the individual transistor information about length and width in the appropriate field of gate's structure. For example, if the two transistors which constitute an inverter have:

1. Length of p equal to 2.
2. Length of n equal to 3.

3. Width of p equal to 4.

4. Width of n equal to 5,

then, a way must be found to include this sizing information in the sizing fields of the identified inverter. This is necessary because, when the isomorphism identification procedure starts, there must be a way to distinguish between inverters of different size.

The formation of the length and width gate labels follows certain rules. These rules are referring to the lengths of p-type transistors; however, the same scheme is applied to n-type transistors or widths. These rules are:

- The length field for each gate is divided into two parts. The first part refers to p-type transistor lengths and the second to n-type transistors. The separation between the values of p-type and n-type transistors is marked by “-”.
- If a circuit consists of more than one p-type transistor, two possibilities exist. Either, all the lengths are the same or they are not. If all lengths are the same, then only one length value appears; otherwise, all lengths appear, separated by “_”.

C. IMPLEMENTATION

A program (see Appendix A) was written in C to implement the gate recognition process. The implementation includes additional steps. These are:

1. Creation of dedicated linked lists, one for every type of gate. These lists point to structures that include the necessary information for each type of gate.
2. Execution of the recognition process for each type of gate.
3. If no gate is found (for a particular type of gate), continue with recognition of another type of gate.

4. Whenever a gate is found, then:

- Perform the length and width placement process, according to the rules described in the previous section of this Chapter.
- Perform recognition of the labels for "input", "output", "phase", and others. This information is needed as the output file will not include information on all transistors. It will include only information about the gate as a circuit.
- Place the necessary information of each gate in a structure. This structure is pointed to by the appropriate linked list.
- Remove the transistors that constitute the recognized gate.

5. Print all gates, with all appropriate information, in the output file.

D. PROGRAM ACCOMPLISHMENTS

The writing and testing of the previously described recognition process completes an important phase of this thesis. The recognition of these *gates demonstrates the possibility and capability to identify more gate structures within a CMOS layout*. However, we have to admit that exhaustive recognition is inefficient in computation time. For example, it may not be needed to search for numerous gates, if there is a small layout in which very few gate types exist.

VI. EXAMINATION OF .SIM FILES FOR UNKNOWN CMOS CIRCUITS.

In Chapter V, it was examined how certain gates can be recognized. The next step is to group the remaining transistors in a useful way. This final task is the examination for existence of isomorphism property. As was stated in Chapter V, whenever a certain gate is recognized, the transistors that constitute this gate are removed from the appropriate linked list.

The remaining transistors in the layout can be divided in two categories [10]. These are:

- Groups of transistors that constitute structures with connection to Vdd and GND, with an existing path between them.
- Groups of transistors that constitute structures without paths to Vdd or GND.

A. STRUCTURES WITH PATH TO VDD AND GND

The algorithm for identification of structures with a path to Vdd and GND requires the following steps:

1. A transistor is selected, with the Vdd label in its source field.
2. A search begins to find all transistors which are connected to the selected transistor's source or drain.
3. A recursive process takes place in order to identify all connected transistors.
4. When a connection to GND is found in any of the connected transistors, the process is considered complete. If no connection to GND is found, there is a possible error.

5. The above process is continued until no remaining transistor exist with connectivity to Vdd and GND within the .sim file.

B. STRUCTURES WITHOUT PATHS TO VDD AND GND

In order to continue with this step, the previous process must be complete. The algorithm for identification of these structures requires the following steps:

1. Select a remaining transistor in the .sim file.
2. Find all transistors which are connected to the selected one.
3. Recursively find all connected transistors.
4. If no other transistor exist in the .sim file, the process is considered complete.

Notice that there are cases in which all steps of the above process are not needed. For example, if we have a single transistor which is not included in a bigger structure, we need only the first two steps.

C. IMPLEMENTATION

The actual implementation of the previously mentioned algorithms includes more steps. This is necessary as managerial procedures have to be added. These procedures play an important role in the correct and efficient execution of the program.

The program first allocates space for a two-dimensional array. This array includes the information needed from these structures. The first field in the array contains the device number. The second field contains pointers to all transistors that constitute the structure. Because these structures (devices) do not have a standard form, we cannot use the same data structure format as before. Redimensioning of the structures and its elements is also necessary to give flexibility. This is important, especially if memory space is limited and a large circuit is examined. However, redimensioning was considered

easier and faster than changing the data structure and parts of the program each time a new circuit is analyzed.

The next step merges the four linked lists into two. The P1 and P2 lists are combined in one, as well as the N1 and N2 lists. This is necessary as abstract devices can have any formation of transistors. The formation must always follow the CMOS general structure, which is, group of P transistors connected to Vdd, followed by N transistors connected to GND.

Then, a search takes place to find a transistor whose gate, source or drain includes the label Vdd. If one (or the first) is found, it is removed from the linked list. This will reduce the execution time, as this transistor will not be selected again.

The search continues along the p-type and n-type transistor lists, to identify transistors that are connected to the chosen one. A recursive search continues until all connected transistors, in the p-type and n-type lists, are reached.

A checking procedure follows, to determine whether or not a connection to GND exist. If no such connection exists, there is a possible error. However, a designer may have transistors within a circuit, which are constantly high or low. This can happen for functional purposes of the circuit. In that case, an error message acts interactively by asking the user whether he wishes to reexamine the .sim file, because no path to GND exists. If the user answers yes, the program is terminated. If the user answers no, the program places this structure as a device and continues its execution.

All transistors that were identified as a structure are placed into an array. The device counter is incremented and the search process starts again. It continues until no other structure with a path from Vdd to GND exists.

The two linked lists with the remaining transistors are combined into a single list. The first transistor of this list is selected and placed on the top of a stack. This transistor is removed from the linked list.

Then recursively compare all transistors which are connected to the selected one. All these transistors are placed on the stack and removed from the linked list. When no other transistor is found which is connected to this device, each transistor of the stack is placed into device array. The algorithm repeats the same process for the current "first" transistor. If no other transistor is left in the single linked list, this phase is considered complete.

VII. IDENTIFICATION FOR ISOMORPHISM.

The isomorphism identification is structured based on the levels discussed in previous Chapters. It is performed at transistor level, gate level and structure level. In each of them, different signatures (properties) are examined and different graph invariants are considered. These differences occur from the variance in information that can be extracted. We now analyze the different factors that are important in isomorphism checking for each level..

A. TRANSISTOR LEVEL

As mentioned in Chapter IV, the transistors that exist in every .sim file are placed in four separate linked lists. This placement was done according to whether or not a connection from Vdd to GND exists. This taxonomy of transistors gives the qualitative property that we are going to use. The number of transistors in each linked list is the needed graph invariant. Four quantitative comparisons are performed between transistors of the two files:

1. Numerical comparison between the P1 list of the one file and the P1 list of the other file.
2. Numerical comparison between the P2 list of the one file and the P2 list of the other file.
3. Numerical comparison between the N1 list of the one file and the N1 list of the other file.
4. Numerical comparison between the N2 list of the one file and the N2 list of the other file.

If the big file has fewer transistors of any type than the small file does, no isomorphism exists between them. If the big file has the same or more, we simply have an indication that isomorphism may exist between them.

B. GATE LEVEL

After the gate recognition process is done to both files, two linked lists exist for each type of gate. Each list contains gates of a given type, one from the big file and another from the small file. The needed signatures come from the taxonomy which has already been performed. The identification, for example, of an inverter implies two transistors (a p-type and a n-type), connected in a certain way as it can be seen in Figure 3.1; the gates connected together, as well as the drains connected together. Source-field of p-type connected to Vdd and source-field of n-type connected to GND.

Furthermore, beside the connectivity implied by this taxonomy, more signatures are tested. The length and the width of each gate were considered to be sufficient signatures. The length and width preserved, as mentioned in Chapter V, the lengths and widths of all transistors that constitute a particular gate. Three comparisons are performed between the lists of a given type gates of the two .sim files:

- Numerical. If, for example, the big file has fewer inverters than the small one, no isomorphism exists.
- Qualitative in lengths. A type of gate may be numerically in order, between the two files, but the gates may differ. They do not differ in type, but in lengths. This is the reason for such a comparison.
- Qualitative in widths. Same comparisons are performed, and for the same reasons, as stated before.

C. STRUCTURE LEVEL

The structure identification process results in the creation of two linked lists, one list for the structures of each file. Isomorphism is verified by examining if a structure which exists in the second file exists also in the first. During this procedure, structures with same total number of transistors are compared. Four types of comparisons are performed:

1. Numerical comparisons between p-type transistors that are connected to Vdd.
2. Numerical comparisons between p-type transistors that are not connected to Vdd.
3. Numerical comparisons between n-type transistors that are connected to GND.
4. Numerical comparisons between n-type transistors that are not connected to GND.

A structure that consists of a set of certain types of transistors must exist in both files in order for isomorphism to be valid.

VIII. CONCLUSIONS.

A. ISOMORPHISM

The program, which is provided within Appendix A, achieves the primary goal of this thesis. This goal was to determine whether examination for isomorphism between two files could be performed. This goal was met through the gate recognition process. Algorithms for each separate type of gate were provided and implemented. This gate recognition process gives the capability for the program to run in two ways, to check for isomorphism or perform device counting. The algorithm's implementation and successful testing verifies the accomplishment of the primary goal.

B. FUTURE RESEARCH

Due to the complexity of VLSI circuits, tools to verify the correctness of a design are considered extremely important. Their importance comes from the additional design time and cost needed to correct design errors. This research gives another tool to the designer to verify the correctness of the design. This research is considered as a part of the initial stage of a much larger VLSI design and timing verifier.

APPENDIX A: PROGRAM LISTINGS

A. GLOBAL VARIABLES

```

/*****
*  EMMANOUIL N. ZAGOURAKIS      THESIS      MAR 1990      *
*  This part of the program contains the common definitions.      *
*****/

#include <string.h>
#include <stdio.h>
#include <malloc.h>

/*****
*  Macro definitions.      *
*****/

#define Function
#define TRUE 1
#define FALSE (!TRUE)
#define IsWhite(x) ((x=='\n') || (x==' ') || (x=='\t'))
#define IsDigit(x) ((060 <= x) && (x <= 071))
#define MAXLEN 50
#define MALLOC(x) ((x *) malloc(sizeof(x)))
#define NUMDEVICE 100
#define NUMTRANS 100
#define ERR_TBL 43

/*****
*  Transistor contains a pointer to the trans type; it also contains pointers *
*  to gate, source, drain, length and width of the transistor; it also contains*
*  a pointer to the next transistor in the linked list.      *
*****/

typedef struct transistor
{
    char *type;
    char *gate;
    char *source;
    char *drain;
    char *length;
    char *width;

```

```

    struct transistor *next;
} trans;
trans *newp1;
trans *newp2;
trans *newn1;
trans *newn2;
trans *curr;
trans *curr1;
trans *(structure[NUMDEVICE][NUMTRANS]);
trans *(structure1[NUMDEVICE][NUMTRANS]);
trans *(stack[NUMTRANS]);
trans *(stack1[NUMTRANS]);

/*****
* Definitions of the pointers in different gate structures.      *
*****/

typedef struct inverter
{
    int flag;
    char *input;
    char *output;
    char *length;
    char *width;
    struct inverter *next;
} inv;

typedef struct prechargedinverter
{
    int flag;
    char *input;
    char *output;
    char *phase;
    char *length;
    char *width;
    struct prechargedinverter *next;
} preinv;

typedef struct i2phclinverter
{
    int flag;
    char *input;
    char *output;

```

```

    char *php;
    char *phn;
    char *length;
    char *width;
    struct i2phclinverter *next;
}i2clinv;

```

```

typedef struct passgate
{
    int flag;
    char *terminal1;
    char *terminal2;
    char *php;
    char *phn;
    char *length;
    char *width;
    struct passgate *next;
}pass;

```

```

typedef struct nor2gate
{
    int flag;
    char *input1;
    char *input2;
    char *output;
    char *length;
    char *width;
    struct nor2gate *next;
}nor2;

```

```

typedef struct nand2gate
{
    int flag;
    char *input1;
    char *input2;
    char *output;
    char *length;
    char *width;
    struct nand2gate *next;
}nand2;

```

```

typedef struct prenand2gate
{
    int flag;
    char *input1;
    char *input2;
    char *phase;
    char *output;
    char *length;
    char *width;
    struct prenand2gate *next;
}prenand2;

```

```

typedef struct prenor2gate
{
    int flag;
    char *input1;
    char *input2;
    char *phase;
    char *output;
    char *length;
    char *width;
    struct prenor2gate *next;
}prenor2;

```

```

typedef struct temporary
{
    char type[MAXLEN+1];
    char gate[MAXLEN+1];
    char source[MAXLEN+1];
    char drain[MAXLEN+1];
    char length[MAXLEN+1];
    char width[MAXLEN+1];
} tem;
tem *temp;

```

```

/*****
* Definitions for head and tail pointers of the different linked lists.  *
*****/

```

```

typedef struct header
{
    int length;
    struct transistor *head, *tail;
}

```

```

} head_type;
head_type *header_new;
head_type *header1_new;
head_type *header_newp;
head_type *header_newn;
head_type *header1_newp;
head_type *header1_newn;
head_type *header_newp1;
head_type *header_newp2;
head_type *header_newn1;
head_type *header_newn2;
head_type *header1_newp1;
head_type *header1_newp2;
head_type *header1_newn1;
head_type *header1_newn2;

```

```

typedef struct headinv
{
    int length;
    struct inverter *head, *tail;
}inve;
inve *head_inv;
inve *head1_inv;

```

```

typedef struct headpreinv
{
    int length;
    struct prechargedinverter *head, *tail;
}preinve;
preinve *head_preinv;
preinve *head1_preinv;

```

```

typedef struct head2clinv
{
    int length;
    struct i2phclinvter *head, *tail;
}i2clinve;
i2clinve *head_2clinv;
i2clinve *head1_2clinv;

```

```

typedef struct headpass
{

```

```

    int length;
    struct passgate *head, *tail;
}passg;
passg *head_pass;
passg *head1_pass;

```

```

typedef struct headnor2
{
    int length;
    struct nor2gate *head, *tail;
}nor2g;
nor2g *head_nor2;
nor2g *head1_nor2;

```

```

typedef struct headnand2
{
    int length;
    struct nand2gate *head, *tail;
}nand2g;
nand2g *head_nand2;
nand2g *head1_nand2;

```

```

typedef struct headprenand2
{
    int length;
    struct prenand2gate *head,*tail;
}prenand2g;
prenand2g *head_prenand2;
prenand2g *head1_prenand2;

```

```

typedef struct headprenor2
{
    int length;
    struct prenor2gate *head, *tail;
}prenor2g;
prenor2g *head_prenor2;
prenor2g *head1_prenor2;

```

```

/*****
* Global variables.
*****/

```

```

FILE *fp;          /* pointer to the input big .sim file      */
FILE *cp;          /* pointer to the input small file */
FILE *fo;          /* pointer to the detailed output file */
FILE *fopen();
FILE *fclose();

char buffer[MAXLEN+1];
char blank[MAXLEN+1];

/*****
* Integer declarations.
*****/

int p1,p2,n1,n2;
int rec;
int totallength;
int totalwidth;
int pcount[NUMDEVICE];
int p1count[NUMDEVICE];
int p2count[NUMDEVICE];
int ncount[NUMDEVICE];
int n1count[NUMDEVICE];
int n2count[NUMDEVICE];
int tcount[NUMDEVICE];
int pcount1[NUMDEVICE];
int p1count1[NUMDEVICE];
int p2count1[NUMDEVICE];
int ncount1[NUMDEVICE];
int n1count1[NUMDEVICE];
int n2count1[NUMDEVICE];
int tcount1[NUMDEVICE];
int numdevice,numdevice1;
int stacknum,stacknum1;
int numtrans,numtrans1;
int ground;
int numn,nump;
int nump1,numn1;

/*****
* External functions.
*****/

extern head_type *create();
extern inve *createinv();

```

```

extern trans *newnode();
extern inv *NewInvert();
extern preinv *NewPreinv();
extern preinve *createpreinv();
extern pass *NewPass();
extern passg *createpass();
extern i2clinv *New2clinv();
extern i2clinve *create2clinv();
extern nor2 *Newnor2();
extern nor2g *createnor2();
extern nand2 *Newnand2();
extern nand2g *createnand2();
extern prenand2 *Newprenand2();
extern prenand2g *createprenand2();
extern prenor2 *Newprenor2();
extern prenor2g *createprenor2();

```

```

/*****
*   Error messages issued by the program.   *
*****/

```

```

static char *msgtbl[ERRTBL] = {
    "Reference file format error in field 1",
    "Reference file format error in field 2, unit:",
    "The system run out of storage space.  ",
    "Reference file format error in field tech:",
    "Reference file format error in filed scmos ",
    "Enter first the big .sim file and then the small .sim file.",
    "No isomorphism! The big file has less p trans. w/o Vdd than the small file.",
    "No isomorphism! The big file has less p trans. with Vdd than the small file.",
    "No isomorphism! The big file has less n trans. w/o GND than the small file.",
    "No isomorphism! The big file has less n trans. with GND than the small file.",
    "No isomorphism! The big file has less inverters than the small file.",
    "No isomorphism! An small file's inverter doesn't match with any of the big. ",
    "No isomorphism! The small file has inverters but the big hasn't.",
    "*****The files are isomorphic in inverter structures.*****",
    "No isomorphism! The big file has less 2ph.clock inverters than the small one.",
    "No isomorphism! A small file's 2ph.clock inverter doesn't match with the big.",
    "No isomorphism! The small file has 2ph. clock inverters but the big hasn't.",
    "*****The files are isomorphic in 2ph. clock inverter structures.****",
    "No isomorphism! The big file has less NOR2 gates than the small file.",
    "No isomorphism! A small file's NOR2 gates doesn't match with any of the big.",
    "No isomorphism! The small file has NOR2 gates but the big hasn't.",
    "*****The files are isomorphic in NOR2 gate structures.*****",
    "No isomorphism! The big file has less NAND2 gates than the small file.",

```

```

"No isomorphism! A small file's NAND2 gates doesn't match with any of the big.",
"No isomorphism! The small file has NAND2 gates but the big hasn't.",
"*****The files are isomorphic in NAND2 gate structures.*****",
"No isomorphism! The big file has less PRECH. NAND2 gates than the small one.",
"No isomorphism! A small file's PRECH. NAND2 gates doesn't match with the big.",
"No isomorphism! The small file has PRECH. NAND2 gates but the big hasn't.",
"*****The files are isomorphic in PRECH. NAND2 gate structures.*****",
"No isomorphism! The big file has less PRECH. NOR2 gates than the small file.",
"No isomorphism! A small file's PRECH. NOR2 gates doesn't match with the big. ",
"No isomorphism! The small file has PRECHARGED NOR2 gates but the big hasn't.",
"*****The files are isomorphic in PRECH. NOR2 gate structures.*****",
"No isomorphism! The big file has less PASSGATES than the small file.",
"No isomorphism! A small file's PASSGATES doesn't match with any of the big. ",
"No isomorphism! The small file has PASSGATES but the big hasn't.",
"*****The files are isomorphic in PASSGATE structures.*****",
"No isomorphism! The big file has less PRECH. inv. gates than the small one.",
"No isomorphism! A small file's PRECH. inv.doesn't match with any of the big. ",
"No isomorphism! The small file has PRECH. inverter gates but the big hasn't.",
"*****The files are isomorphic in PRECH. inverter structures.****",
"*****The files are isomorphic in number of transistors. *****"
};

```

```

/*****
* This program takes as input 2 files in .sim format and tries to find *
* if isomorphism exists between the two files. This is done by examining *
* the existance of certain gate structures and by comparing the number *
* of each kind of gate. If we consider the first file bigger than the *
* second it won't be isomorphism if eg. the small file has more inverters*
* than the big file. *
* It can also be used as a verifier of VLSI layouts if we know in advance*
* of the test how many gates of a certain kind exist in the layout. In *
* case a mistake of any kind may have occured in a certain gate,the total*
* number of gates of that kind will be less than expected. *
* The results exist in two files. The first is the "output" file which *
* gives us more details about each gate which was identified (such as *
* input,output,phase e.t.c.). The second is the "isooutput" which give us*
* only the number of gates identified by the program, indexed by file and*
* by kind of gate. *
*****/

```

```

#include "headers.h"

```

```

Function main(argc,argv)

int argc;
char **argv;
{
    int refok,simok;
    int rec;

    refok=simok=FALSE;
    rec=0;
    strcpy(blank,"");

    if ((argc > 3)|| (argc==1)) {
        error(5);
        exit(1);
    }
    if(argc==2) rec=1;

    create_head(); /* for header nodes */

    fp = fopen(argv[1],"r"); /* file from the big layout */
    fo = fopen("output","w");

    strcpy(buffer,blank);
    refok = proper(fp);
    if (refok == TRUE) {
        fprintf(fo,"***** F I R S T   file   (Large one) *****\n");
        temp=MALLOC(tem);
        transistor(fp);
        printtrans();
        p1=header_newp1->length;
        p2=header_newp2->length;
        n1=header_newn1->length;
        n2=header_newn2->length;

        compareinv();
        print_inv();
        head1_inv=head_inv;
        head_inv=createinv();

        comparepass();
        print_pass();
        head1_pass=head_pass;
        head_pass=createpass();
    }
}

```

```

compare2clinv();
print_2clinv();
head1_2clinv=head_2clinv;
head_2clinv=create2clinv();

comparenor2();
print_nor2();
head1_nor2=head_nor2;
head_nor2=createnor2();

comparenand2();
print_nand2();
head1_nand2=head_nand2;
head_nand2=createnand2();

compareprecharged();
print_prenand2();
print_prenor2();
head1_prenand2=head_prenand2;
head_prenand2=createprenand2();
head1_prenor2=head_prenor2;
head_prenor2=createprenor2();

comparepreinv();
print_preinv();
head1_preinv=head_preinv;
head_preinv=createpreinv();
/*****
*   Merge two lists p1 and p2.
*****/
header1_newp->length = header_newp2->length + header_newp1->length;
header1_newn->length = header_newn2->length + header_newn1->length;
if(header_newp1->length ==0) {
    (header_newp1->head) = header_newp2->head;
    (header_newp1->tail) = header_newp2->tail;}
else {
    (header_newp1->tail)->next = header_newp2->head;
}

header1_newp->head = header_newp1->head;
header1_newp->tail = header_newp2->tail;

if(header_newn1->length ==0) {
    (header_newn1->head) = header_newn2->head;
    (header_newn1->tail) = header_newn2->tail;}

```

```

else {
    (header_newn1->tail)->next = header_newn2->head;
}
header1_newn->head = header_newn1->head;
header1_newn->tail = header_newn2->tail;
header_newp1->length=header_newp2->length=0;
header_newn1->length=header_newn2->length=0;

comparestructures1();
print_structures1();

if (argc==3) {
    cp = fopen(argv[2],"r");    /* file from the reference layout */
    simok=proper(cp);
}
else exit(0);
fprintf(fo,"***** S E C O N D   file (Small one) *****\n");
if((simok == TRUE)&&(rec!=1)) {    /* rec==1 when we have one file */
    strcpy(buffer,blank);

    transistor(cp);    /* create trans lists */
    printtrans();    /* print trans lists */
    isotrans();

    compareinv();
    print_inv();
    if((head1_inv->head!=NULL)&&(head_inv->head!=NULL)) isoinv();
    else if((head_inv->head!=NULL)&&(head1_inv->head==NULL)) error(12);
    head_inv=createinv();

    comparepass();
    print_pass();
    if((head1_pass->head!=NULL)&&(head_pass->head!=NULL)) isopass();
    else if((head_pass->head!=NULL)&&(head1_pass->head==NULL))
        error(36);
    head_pass=createpass();

    compare2clinv();
    print_2clinv();
    if((head1_2clinv->head!=NULL)&&(head_2clinv!=NULL))
        iso2clinv();
    else if((head1_2clinv->head!=NULL)&&(head_2clinv->head==NULL))
        error(16);
    head_2clinv=create2clinv();
}

```

```

        comparenor2();
        print_nor2();
        if((head1_nor2->head!=NULL)&&(head_nor2->head!=NULL)) isonor2();
        else if((head_nor2->head!=NULL)&&(head1_nor2->head==NULL)) error(20);
        head_nor2=createnor2();

        comparenand2();
        print_nand2();
        if((head1_nand2->head!=NULL)&&(head_nand2->head!=NULL)) isonand2();
        else if((head_nand2->head!=NULL)&&(head1_nand2->head==NULL))
            error(24);
        head_nand2=createnand2();

        compareprecharged();
        print_prenand2();
        print_prenor2();
        if((head1_prenand2->head!=NULL)&&(head_prenand2->head!=NULL))
            isoprenand2();
        else if((head_prenand2->head!=NULL)&&(head1_prenand2->head==NULL))
            error(28);
        head_prenand2=createprenand2();
        if((head1_prenor2->head!=NULL)&&(head_prenor2->head!=NULL))
            isoprenor2();
        else if((head_prenor2->head!=NULL)&&(head1_prenor2->head==NULL))
            error(32);
        head_prenor2=createprenor2();

        comparepreinv();
        print_preinv();
        if((head1_preinv->head!=NULL)&&(head_preinv->head!=NULL))
            isopreinv();
        else if((head_preinv->head!=NULL)&&(head1_preinv->head==NULL))
            error(40);
        head_preinv=createpreinv();

        comparestructures();
        print_structures();
        isostructures();
    }
}
fclose(fp);
fclose(cp);
fclose(fo);
}

```

```

/*****
* This function clears structures of transistors.
*****/

```

```

Function trans *clean(tranptr)
trans *tranptr;
{
    strcpy(tranptr->type, " ");
    strcpy(tranptr->gate, blank);
    strcpy(tranptr->source, blank);
    strcpy(tranptr->drain, blank);
    strcpy(tranptr->length, blank);
    strcpy(tranptr->width, blank);
    tranptr->next = NULL;
}

```

```

/*****
* This function removes all elements of a linked list.
*****/

```

```

Function cleanlist(first)
trans *first;
{
    trans *current, *next;
    current = first;
    while(current != NULL){ next = current->next;
                           free(current);
                           current = next;
    }
}

```

```

/*****
* This function builds indirectly the 4 transistor lists which are:
* 1. p transistors with Vdd in either source or drain fields
* 2. p transistors w/o Vdd in either source or drain fields
* 3. n transistors with GND in either source or drain fields
* 4. n transistors w/o GND in either source or drain fields
*****/

```

```

Function transistor(p)
FILE *p;
{

```

```

while(TRUE) {
    fscanf(p,"%s",buffer);
    if((strcmp(buffer,"p")==0) {
        strcpy(temp->type,buffer);
        ptransistor(p);    /* put in appropriate list */
    }
    else if((strcmp(buffer,"n")==0) {
        strcpy(temp->type,buffer);
        ntransistor(p);
    }
    else break;
} /* end while */
}

/*****
 * This function builds the 2 link lists of p type transistors.      *
 * It also places the "Vdd" label in source field in case it exists in the *
 * drain field, in order to simplify all later comparisons between transistors*
 * which are needed to identify all kind of gates.                  *
 *****/

Function ptransistor(p)
FILE *p;

{
    trans *curr1,*curr2;
    int done;

    curr1=header_newp1->tail;    /* p1 not connecting to Vdd */
    curr2=header_newp2->tail;    /* p2 are those connect to Vdd */
    done=FALSE;

    while(!done) {
        fscanf(p,"%s %s %s ",temp->gate,temp->source,temp->drain);
        fscanf(p,"%s %s ",temp->length,temp->width);
        fscanf(p,"%s",buffer);
        fscanf(p,"%s",buffer);

        if(((strcmp(temp->source,"Vdd")==0)||((strcmp(temp->drain,"Vdd")==0)))
        {
            header_newp2->length++;
            newp2 = newnode();
            newp2->type=malloc(sizeof(char) * strlen(temp->type)+1);
            newp2->gate=malloc(sizeof(char) * strlen(temp->gate)+1);

```

```

newp2->length=malloc(sizeof(char) * strlen(temp->length)+1);
newp2->width=malloc(sizeof(char) * strlen(temp->width)+1);

strcpy(newp2->type,temp->type);
strcpy(newp2->gate,temp->gate);
strcpy(newp2->length,temp->length);
strcpy(newp2->width,temp->width);

if((strcmp(temp->source,"Vdd"))==0) {
    newp2->source=malloc(sizeof(char) * strlen(temp->source)+1);
    newp2->drain=malloc(sizeof(char) * strlen(temp->drain)+1);

    strcpy(newp2->source,temp->source);
    strcpy(newp2->drain,temp->drain);
}

else {
    newp2->source=malloc(sizeof(char) * strlen(temp->drain)+1);
    newp2->drain=malloc(sizeof(char) * strlen(temp->source)+1);
    strcpy(newp2->source,temp->drain);
    strcpy(newp2->drain,temp->source);
} /*end else */

/*****
*   Is this the first entry in the list?
*****/
    if(header_newp2->head == NULL) {
        header_newp2->head=newp2;
        header_newp2->tail=newp2;
    }

    else {
        header_newp2->tail=newp2;
        curr2->next=newp2;
    }
}

else {
    neader_newp1->length++;
    newp1=newnode();
    if(header_newp1->head==NULL) {
        header_newp1->head=newp1;
        header_newp1->tail=newp1;
    }
    else {
        header_newp1->tail=newp1;
        curr1->next=newp1;
    }
}

```

```

newp1->type=malloc(sizeof(char) * strlen(temp->type)+1);
newp1->gate=malloc(sizeof(char) * strlen(temp->gate)+1);
newp1->source=malloc(sizeof(char) * strlen(temp->source)+1);
newp1->drain=malloc(sizeof(char) * strlen(temp->drain)+1);
newp1->length=malloc(sizeof(char) * strlen(temp->length)+1);
newp1->width=malloc(sizeof(char) * strlen(temp->width)+1);

strcpy(newp1->type,"p");
strcpy(newp1->gate,temp->gate);
strcpy(newp1->source,temp->source);
strcpy(newp1->drain,temp->drain);
strcpy(newp1->length,temp->length);
strcpy(newp1->width,temp->width);
}
done=TRUE;
}
}

```

```

/*****
* This function builds the 2 link lists for the n type transistors.      *
* It also places "GND" label in the source field in case it is in the drain*
* field.                                                                    *
*****/

```

Function ntransistor(p)

FILE *p;

```

{
trans *curr1,*curr2;
int done;
curr1=header_newn1->tail;
curr2=header_newn2->tail;
done=FALSE;
while(!done) {
fscanf(p,"%s",temp->gate);
fscanf(p,"%s",temp->source);
fscanf(p,"%s",temp->drain);
fscanf(p,"%s",temp->length);
fscanf(p,"%s",temp->width);
fscanf(p,"%s",buffer);
fscanf(p,"%s",buffer);

if(((strcmp(temp->source,"GND"))==0)||((strcmp(temp->drain,"GND"))==0))

```

```

{
    header_newn2->length++;
    newn2 = newnode();
    newn2->type=malloc(sizeof(char) * strlen(temp->type)+1);
    newn2->gate=malloc(sizeof(char) * strlen(temp->gate)+1);
    newn2->length=malloc(sizeof(char) * strlen(temp->length)+1);
    newn2->width=malloc(sizeof(char) * strlen(temp->width)+1);

    strcpy(newn2->type,temp->type);
    strcpy(newn2->gate,temp->gate);
    strcpy(newn2->length,temp->length);
    strcpy(newn2->width,temp->width);

    if((strcmp(temp->source,"GND")==0) {
        newn2->source=malloc(sizeof(char) * strlen(temp->source)+1);
        newn2->drain=malloc(sizeof(char) * strlen(temp->drain)+1);
        strcpy(newn2->drain,temp->drain);
        strcpy(newn2->source,temp->source);
    }

    else {
        newn2->source=malloc(sizeof(char) * strlen(temp->drain)+1);
        newn2->drain=malloc(sizeof(char) * strlen(temp->source)+1);
        strcpy(newn2->source,temp->drain);
        strcpy(newn2->drain,temp->source);
    } /* end else */
}

/*****
*   Is this the first entry in the list?
*****/
    if(header_newn2->head == NULL) {
        header_newn2->head=newn2;
        header_newn2->tail=newn2;
    }
    else {
        header_newn2->tail=newn2;
        curr2->next=newn2;
    }
}

else {
    header_newn1->length++;
    newn1=newnode();
    if(header_newn1->head==NULL) {
        header_newn1->head=newn1;
        header_newn1->tail=newn1;
    }
}

```

```

    }

    else {
        header_newn1->tail=newn1;
        curri->next=newn1;
    }
    newn1->type=malloc(sizeof(char) * strlen(temp->type)+1);
    newn1->gate=malloc(sizeof(char) * strlen(temp->gate)+1);
    newn1->source=malloc(sizeof(char) * strlen(temp->source)+1);
    newn1->drain=malloc(sizeof(char) * strlen(temp->drain)+1);
    newn1->length=malloc(sizeof(char) * strlen(temp->length)+1);
    newn1->width=malloc(sizeof(char) * strlen(temp->width)+1);
    strcpy(newn1->type,temp->type);
    strcpy(newn1->gate,temp->gate);
    strcpy(newn1->source,temp->source);
    strcpy(newn1->drain,temp->drain);
    strcpy(newn1->length,temp->length);
    strcpy(newn1->width,temp->width);
}
done=TRUE;
}
}

```

B. GATE RECOGNITION PROCESS

```

/*****
 * This function identifies the inverters and places them in a linked list *
 *****/

#include "headers.h"

Function compareinv()
{
    int done;
    int complete;
    int stop;

    int totallength;
    int totalwidth;

    trans *first,*second;
    trans *prev1,*prev2;
    inv *curr;
    inv *newinv;

    first=header_newp2->head;
    second=header_newn2->head;
    prev1=first;
    prev2=second;

    totallength=totalwidth=0;

    stop=FALSE;

    while((first!=NULL)&&(stop==FALSE)) { /* search through the transistor */
        complete=FALSE;
        done=FALSE;
        while((second!=NULL)&&(!done)) {
            if(((strcmp(first->gate,second->gate))==0)&& /* is it an inverter? */
                ((strcmp(first->drain,second->drain))==0)) {
                head_inv->length++; /* increment the length of link list for
                                     printing purposes */
                curr=head_inv->tail;
                done=TRUE;
                newinv=NewInvert(); /* create a node in the link list to hold
                                     the informations about the found
                                     inverter */
                if(head_inv->head==NULL){ /* is this the first entry in the list? */

```

```

        head_inv->head=newinv;
        head_inv->tail=newinv;
    }
    else {
        /* if this is not the first entry it is
           the last entry */
        head_inv->tail=newinv;
        curr->next=newinv;
    }

    /*****
    *   allocate needed space for the appropriate data
    *****/
    newinv->input=malloc(sizeof(char) * strlen(first->gate)+1);
    newinv->output=malloc(sizeof(char) * strlen(first->drain)+1);

    /*****
    *   place the data
    *****/
    strcpy(newinv->input,first->gate);
    strcpy(newinv->output,first->drain);
    if((strcmp(first->length,second->length))==0) {
        newinv->length=malloc(sizeof(char)* strlen(first->length)+1);
        strcpy(newinv->length,first->length);
    }
    else {
        totallength=(strlen(first->length)+strlen(second->length)+2);
        newinv->length=malloc(sizeof(char)* (totallength)+1);
        strcpy(newinv->length,first->length);
        strcat(newinv->length,"--");
        strcat(newinv->length,second->length);
    }
    if((strcmp(first->width,second->width))==0) {
        newinv->width=malloc(sizeof(char)* strlen(first->width)+1);
        strcpy(newinv->width,first->width);
    }
    else {
        totalwidth=(strlen(first->width)+strlen(second->width)+2);
        newinv->width=malloc(sizeof(char) * (totalwidth)+1);
        strcpy(newinv->width,first->width);
        strcat(newinv->width,"--");
        strcat(newinv->width,second->width);
    }
}

    /*****
    * Remove the transistors that now constitute an inverter. Delete the used*
    * transistors and decrease the transistors lists' length.
    *****/
    if(header_newn2->tail==second) {

```

```

        header_newn2->tail=prev2;
        prev2->next=NULL;
        complete=TRUE;
    }
    if(header_newn2->head==second) {
        header_newn2->head=second->next;
    }
    if((second!=prev2)&&(complete!=TRUE)) {
        prev2->next=prev2->next->next;
        second=second->next;
    }
    else {
        prev2=prev2->next;
        second=second->next;
    }
    if(header_newp2->tail==first) {
        stop=TRUE;
        header_newp2->tail=prev1;
        prev1->next=NULL;
    }
    if (header_newp2->head==first) {
        header_newp2->head=first->next;
    }
    if((first!=prev1)&&(stop!=TRUE)) {
        prev1->next=prev1->next->next;
        first=first->next;
    }
    else {
        prev1=prev1->next;
        first=first->next;
    }
    header_newp2->length--;
    header_newn2->length--;
    curr=head_inv->tail;
}
else {
    prev2=second;
    second=second->next;
}
}
if((header_newp2->length==0)&&(header_newn2->length==0)) break;
if(done!=TRUE) {
    prev1=first;
    first=first->next;
}
}

```

```

        second=header_newn2->head;
        prev2=second;
    }
}

```

```

/*****
*   This function identifies the precharged inverters and places in a link*
*   list.
*****/

```

```

Function comparepreinv()
{

```

```

    int end;
    int done;
    int complete;
    int stop;

```

```

    trans *first, *second, *third;
    trans *prev1, *prev2, *prev3;
    preinv *curri;
    preinv *newpreinv;

```

```

    first=header_newp2->head;
    second=header_newn1->head;
    third=header_newn2->head;

```

```

    prev1=first;
    prev2=second;
    prev3=third;

```

```

    totallength=totalwidth=0;
    stop=FALSE;

```

```

    while((first !=NULL)&&(stop == FALSE)) { /* search through the transistor
                                                list to find a precharged inv.*/
        complete=FALSE;
        done=FALSE;
        while((second !=NULL)&&(!done)) {
            end=FALSE;
            while((third!=NULL)&&(!end)) {
                if(((strcmp(first->gate,third->gate))==0)&& /* is it a prech. inv.?*/
                    ((strcmp(first->gate,second->gate))!=0)&&
                    ((strcmp(first->gate,first->drain))!=0)&&

```

```

((strcmp(first->gate,third->drain))!=0)&&
((strcmp(second->gate,second->drain))!=0)&&
((((strcmp(first->drain,second->source))==0)&&
((strcmp(second->drain,third->drain))==0))||
(((strcmp(first->drain,second->drain))==0)&&
((strcmp(second->source,third->drain))==0)))) {
head_preinv->length++;    /* increment the length of link list for
                           printing purposes */
curr1=head_preinv->tail;
end=TRUE;
newpreinv=NewPreinv();    /* create a node in the link list to hold
                           the informations about the found prech.
                           inverter. */
if(head_preinv->head==NULL){ /*is this the first entry in the list?*/
head_preinv->head=newpreinv;
head_preinv->tail=newpreinv;
}
else {                    /* if this is not the first entry it is
                           the last entry. */
head_preinv->tail=newpreinv;
curr1->next=newpreinv;
}
newpreinv->input=malloc(sizeof(char) * strlen(first->gate)+1);
newpreinv->output=malloc(sizeof(char) * strlen(first->drain)+1);
newpreinv->phase=malloc(sizeof(char) * strlen(second->gate)+1);
strcpy(newpreinv->input,first->gate);    /* place the data. */
strcpy(newpreinv->output,first->drain);
strcpy(newpreinv->phase,second->gate);
if((((strcmp(first->length,second->length))==0)&&
((strcmp(second->length,third->length))==0)) {
newpreinv->length=malloc(sizeof(char)* strlen(first->length)+1);
strcpy(newpreinv->length,first->length);
}
else if ((strcmp(second->length,third->length))==0) {
totallength=(strlen(first->length)+strlen(second->length)+2);
newpreinv->length=malloc(sizeof(char)* (totallength)+1);
strcpy(newpreinv->length,first->length);
strcat(newpreinv->length,"--");
strcat(newpreinv->length,second->length);
}
else {
totallength=(strlen(first->length)+strlen(second->length));
totallength=(strlen(third->length)+totallength+3);
newpreinv->length=malloc(sizeof(char)* (totallength)+1);
strcpy(newpreinv->length,first->length);

```

```

    strcat(newpreinv->length,"--");
    strcat(newpreinv->length,second->length);
    strcat(newpreinv->length,"-");
    strcat(newpreinv->length,third->length);
}
if(((strcmp(first->width,second->width))==0)&&
    ((strcmp(second->width,third->width))==0)) {
    newpreinv->width=malloc(sizeof(char)* strlen(first->width)+1);
    strcpy(newpreinv->width,first->width);
}
else if((strcmp(second->width,third->width))==0) {
    totalwidth=(strlen(second->width)+strlen(third->width)+2);
    newpreinv->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newpreinv->width,first->width);
    strcat(newpreinv->width,"--");
    strcat(newpreinv->width,second->width);
}
else {
    totalwidth=(strlen(first->width)+strlen(second->width));
    totalwidth=(strlen(third->width)+totalwidth+3);
    newpreinv->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newpreinv->width,first->width);
    strcat(newpreinv->width,"--");
    strcat(newpreinv->width,second->width);
    strcat(newpreinv->width,"-");
    strcat(newpreinv->width,third->width);
}
}
/*****
* Remove the transistors that now constitute an precharged inveter. Delete *
* the used transistors and decrease the transistors lists' length      *
*****/
    if(header_newn2->tail==third) {
        header_newn2->tail=prev3;
        prev3->next=NULL;
        complete=TRUE;
    }
    if(header_newn2->head==third) {
        header_newn2->head=third->next;
    }
    if((third!=prev3)&&(complete!=TRUE)) {
        prev3->next=prev3->next->next;
        third=third->next;
    }
    else {
        prev3=prev3->next;

```

```

        third=third->next;
    }
    if(header_newn1->tail==second) {
        header_newn1->tail=prev2;
        prev2->next=NULL;
        done=TRUE;
    }
    if(header_newn1->head==second) {
        header_newn1->head=second->next;
    }
    if((second!=prev2)&&(end!=TRUE)) {
        prev2->next=prev2->next->next;
        second=second->next;
    }
    else {
        prev2=prev2->next;
        second=second->next;
    }
    if(header_newp2->tail==first) {
        header_newp2->tail=prev1;
        prev1->next=NULL;
        stop=TRUE;
    }
    if(header_newp2->head==first) {
        header_newp2->head=first->next;
    }
    if((first!=prev1)&&(done!=TRUE)) {
        prev1->next=prev1->next->next;
        first=first->next;
    }
    else {
        prev1=prev1->next;
        first=first->next;
    }
    header_newp2->length--;
    header_newn1->length--;
    header_newn2->length--;
    curr1=head_preinv->tail;
}
else {
    prev3=third;
    third=third->next;
}
if(header_newn2->length==0)break;
}

```

```

        if(header_newn1->length==0)break;
        if(end!=TRUE) {
            prev2=second;
            second=second->next;
        }
        third=header_newn2->head;
        prev3=third;
    }
    if(header_newp2->length==0)break;
    if(done!=TRUE) {
        prev1=first;
        first=first->next;
    }
    second=header_newn1->head;
    prev2=second;
    third=header_newn2->head;
    prev3=third;
}
}

```

```

/*****
* This function identifies the passgates and places them in a link list.  *
*****/

```

```

Function comparepass()
{
    int done;
    int complete;
    int stop;

    int totallength;
    int totalwidth;

    trans *first, *second;
    trans *prev1, *prev2;

    pass *curr;
    pass *newpass;

    first=header_newp1->head;
    second=header_newn1->head;
    prev1=first;
    prev2=second;

```

```

totallength=totalwidth=0;

stop=FALSE;

while((first!=NULL)&&(stop==FALSE)) {
    complete=FALSE;
    done=FALSE;
    while((second!=NULL)&&(!done)) {
        if((strcmp(first->gate,second->gate)!=0)&&
            (((strcmp(first->source,second->source)==0)&&
              (strcmp(first->drain,second->drain)==0))||
              ((strcmp(first->source,second->drain)==0)&&
                (strcmp(first->drain,second->source)==0)))))) {
            head_pass->length++;
            curr=head_pass->tail;
            done=TRUE;
            newpass=NewPass();

            if(head_pass->head==NULL) {
                head_pass->head=newpass;
                head_pass->tail=newpass;
            }
            else {
                head_pass->tail=newpass;
                curr->next=newpass;
            }

            newpass->terminal1=malloc(sizeof(char)*strlen(first->source)+1);
            newpass->terminal2=malloc(sizeof(char)*strlen(first->drain)+1);
            newpass->php=malloc(sizeof(char)*strlen(first->gate)+1);
            newpass->phn=malloc(sizeof(char)*strlen(second->gate)+1);

            strcpy(newpass->terminal1,first->source);
            strcpy(newpass->terminal2,first->drain);
            strcpy(newpass->php,first->gate);
            strcpy(newpass->phn,second->gate);
            if((strcmp(first->length,second->length))==0) {
                newpass->length=malloc(sizeof(char)*strlen(first->length)+1);
                strcpy(newpass->length,first->length);
            }
            else {
                totallength=(strlen(first->length)+strlen(second->length)+2);
                newpass->length=malloc(sizeof(char)*(totallength)+1);
                strcpy(newpass->length,first->length);
                strcat(newpass->length,"--");
            }
        }
    }
}

```

```

    strcat(newpass->length,second->length);
}
if((strcmp(first->width,second->width))==0) {
    newpass->width=malloc(sizeof(char)*strlen(first->width)+1);
    strcpy(newpass->width,first->width);
}
else {
    totalwidth=(strlen(first->width)+strlen(second->width)+2);
    newpass->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newpass->width,first->width);
    strcat(newpass->width,"--");
    strcat(newpass->width,second->width);
}
if(header_newn1->tail==second) {
    header_newn1->tail=prev2;
    prev2->next=NULL;
    complete=TRUE;
}
if(header_newn1->head==second) {
    header_newn1->head=second->next;
}
if((second!=prev2)&&(complete!=TRUE)) {
    prev2->next=prev2->next->next;
    second=second->next;
}
else {
    prev2=prev2->next;
    second=second->next;
}
if(header_newp1->tail==first) {
    stop=TRUE;
    header_newp1->tail=prev1;
    prev1->next=NULL;
}
if(header_newp1->head==first) {
    header_newp1->head=first->next;
}
if((first!=prev1)&&(stop!=TRUE)) {
    prev1->next=prev1->next->next;
    first=first->next;
}
else {
    prev1=prev1->next;
    first=first->next;
}
}

```

```

        header_newp1->length--;
        header_newn1->length--;
        curr=head_pass->tail;
    }
    else {
        prev2=second;
        second=second->next;
    }
}
if((header_newp1->length==0)|| (header_newn1->length==0)) break;
if(done!=TRUE) {
    prev1=first;
    first=first->next;
}
second=header_newn1->head;
prev2=second;
}
}

```

```

/*****
* This function identifies the 2 phase locked inverters and places them
* in alink list.
*****/

```

```

Function compare2clinv()
{
    int finished;
    int end;
    int done;
    int complete;
    int stop;

    trans *first, *second, *third, *forth;
    trans *prev1, *prev2, *prev3, *prev4;
    i2clinv *curr;
    i2clinv *new2clinv;

    first=header_newp1->head;
    second=header_newp2->head;
    third=header_newn1->head;
    forth=header_newn2->head;

```

```

prev1=first;
prev2=second;
prev3=third;
prev4=forth;

totallength=totalwidth=0;
stop=FALSE;

while((first !=NULL)&&(stop == FALSE)) {
    complete=FALSE;
    done=FALSE;
    while((second !=NULL)&&(!done)) {
        end=FALSE;
        while((third !=NULL)&&(!end)) {
            finished=FALSE;
            while((forth !=NULL)&&(!finished)) {
                if((((strcmp(first->gate,third->gate))==0)&&
                    ((strcmp(second->gate,forth->gate))!=0)&&
                    ((strcmp(second->gate,first->gate))!=0)) ||
                    ((strcmp(first->gate,second->gate))!=0)&&
                    ((strcmp(first->gate,third->gate))!=0)&&
                    ((strcmp(second->gate,forth->gate))==0))) &&
                    (((strcmp(second->drain,first->source))==0)&&
                    ((strcmp(first->drain,third->drain))==0)&&
                    ((strcmp(third->source,forth->drain))==0)) ||
                    ((strcmp(first->drain,second->drain))==0)&&
                    ((strcmp(first->source,third->drain))==0)&&
                    ((strcmp(third->source,forth->drain))==0)) ||
                    ((strcmp(first->source,second->drain))==0)&&
                    ((strcmp(first->drain,third->source))==0)&&
                    ((strcmp(third->drain,forth->drain))==0)) ||
                    (((strcmp(first->drain,second->drain))==0)&&
                    ((strcmp(first->source,third->source))==0)&&
                    ((strcmp(third->drain,forth->drain))==0)))) {
                    head_2clinv->length++;
                    curr=head_2clinv->tail;
                    finished=TRUE;
                    new2clinv=New2clinv();
                    if(head_2clinv->head == NULL) {
                        head_2clinv->head=new2clinv;
                        head_2clinv->tail=new2clinv;
                    }
                    else {
                        head_2clinv->tail=new2clinv;
                        curr->next=new2clinv;
                    }
                }
            }
        }
    }
}

```

```

}
if((strcmp(first->gate,third->gate))==0) {
    new2clinv->input=malloc(sizeof(char)*strlen(second->gate)+1);
    new2clinv->php=malloc(sizeof(char)*strlen(first->gate)+1);
    new2clinv->phn=malloc(sizeof(char)*strlen(forth->gate)+1);

    strcpy(new2clinv->input,second->gate);
    strcpy(new2clinv->php,first->gate);
    strcpy(new2clinv->phn,forth->gate);
    if(((strcmp(first->drain,third->drain))==0)||
        ((strcmp(first->drain,third->source))==0)) {
        new2clinv->output=malloc(sizeof(char)*strlen(first->drain)+1);
        strcpy(new2clinv->output,first->drain);
    }
    else {
        new2clinv->output=malloc(sizeof(char)*strlen(first->source)+1);
        strcpy(new2clinv->output,first->source);
    }
}
else {
    new2clinv->input=malloc(sizeof(char)*strlen(first->gate)+1);
    new2clinv->php=malloc(sizeof(char)*strlen(second->gate)+1);
    new2clinv->phn=malloc(sizeof(char)*strlen(third->gate)+1);

    strcpy(new2clinv->input,first->gate);
    strcpy(new2clinv->php,second->gate);
    strcpy(new2clinv->phn,third->gate);

    if(((strcmp(first->drain,third->drain))==0)||
        ((strcmp(first->drain,third->source))==0)) {
        new2clinv->output=malloc(sizeof(char)*strlen(first->drain)+1);
        strcpy(new2clinv->output,first->drain);
    }
    else {
        new2clinv->output=malloc(sizeof(char)*strlen(first->source)+1);
        strcpy(new2clinv->output,first->source);
    }
}
if(((strcmp(first->length,second->length))==0)&&
    ((strcmp(first->length,forth->length))==0)&&
    ((strcmp(second->length,third->length))==0)) {
    new2clinv->length=malloc(sizeof(char)*strlen(first->length)+1);
    strcpy(new2clinv->length,first->length);
}
else if (((strcmp(second->length,first->length))==0)&&

```

```

        ((strcmp(third->length,forth->length))!=0)) {
    totallength=(strlen(first->length)+strlen(third->length)+3);
    totallength=(strlen(forth->length)+totallength);
    new2clinv->length=malloc(sizeof(char)*(totallength)+1);

    strcpy(new2clinv->length,first->length);
    strcat(new2clinv->length,"--");
    strcat(new2clinv->length,third->length);
    strcat(new2clinv->length,"-");
    strcat(new2clinv->length,forth->length);
}
else if(((strcmp(first->length,second->length))!=0)&&
        ((strcmp(third->length,forth->length))==0)) {
    totallength=(strlen(first->length)+strlen(second->length)+3);
    totallength=(strlen(third->length)+totallength);
    new2clinv->length=malloc(sizeof(char)*(totallength)+1);

    strcpy(new2clinv->length,first->length);
    strcat(new2clinv->length,"-");
    strcat(new2clinv->length,second->length);
    strcat(new2clinv->length,"--");
    strcat(new2clinv->length,third->length);
}
else {
    totallength=(strlen(first->length)+strlen(second->length));
    totallength=(strlen(third->length)+totallength+4);
    totallength=(totallength+strlen(forth->length));
    new2clinv->length=malloc(sizeof(char)*(totallength)+1);

    strcpy(new2clinv->length,first->length);
    strcat(new2clinv->length,"-");
    strcat(new2clinv->length,second->length);
    strcat(new2clinv->length,"--");
    strcat(new2clinv->length,third->length);
    strcat(new2clinv->length,"-");
    strcat(new2clinv->length,forth->length);
}
if(((strcmp(first->width,second->width))==0)&&
    ((strcmp(first->width,forth->width))==0)&&
    ((strcmp(second->width,third->width))==0)) {
    new2clinv->width=malloc(sizeof(char)*strlen(first->width)+1);
    strcpy(new2clinv->width,first->width);
}
else if(((strcmp(second->width,first->width))==0)&&
        ((strcmp(third->width,forth->width))!=0)) {

```

```

totalwidth=(strlen(first->width)+strlen(third->width)+3);
totalwidth=(totalwidth+strlen(forth->width));
new2clinv->width=malloc(sizeof(char)*(totalwidth)+1);

strcpy(new2clinv->width,first->width);
strcat(new2clinv->width,"--");
strcat(new2clinv->width,third->width);
strcat(new2clinv->width,"-");
strcat(new2clinv->width,forth->width);
}
else if(((strcmp(first->width,second->width))!=0)&&
        ((strcmp(third->width,forth->width))==0)) {
    totalwidth=(strlen(first->width)+strlen(second->width));
    totalwidth=(totalwidth+strlen(third->width)+3);
    new2clinv->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(new2clinv->width,first->width);
    strcat(new2clinv->width,"-");
    strcat(new2clinv->width,second->width);
    strcat(new2clinv->width,"--");
    strcat(new2clinv->width,third->width);
}
else {
    totalwidth=(strlen(second->width)+strlen(third->width));
    totalwidth=(totalwidth+strlen(first->width)+strlen(forth->width)+4);
    new2clinv->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(new2clinv->width,first->width);

    strcat(new2clinv->width,"-");
    strcat(new2clinv->width,second->width);
    strcat(new2clinv->width,"--");
    strcat(new2clinv->width,third->width);
    strcat(new2clinv->width,"-");
    strcat(new2clinv->width,forth->width);
}
if(header_newn2->tail==forth) {
    header_newn2->tail=prev4;
    prev4->next=NULL;
    complete=TRUE;
}
if(header_newn2->head==forth) {
    header_newn2->head=forth->next;
}
if((forth!=prev4)&&(complete!=TRUE)) {
    prev4->next=prev4->next->next;
    forth=forth->next;
}

```

```

    }
    else {
        prev4=prev4->next;
        forth=forth->next;
    }
    if(header_newn1->tail==third) {
        header_newn1->tail=prev3;
        prev3->next=NULL;
        end=TRUE;
    }
    if(header_newn1->head==third) {
        header_newn1->head=third->next;
    }
    if((third!=prev3)&&(finished!=TRUE)) {
        prev3->next=prev3->next->next;
        third=third->next;
    }
    else {
        prev3=prev3->next;
        third=third->next;
    }
    if(header_newp2->tail==second) {
        header_newp2->tail=prev2;
        prev2->next=NULL;
        done=TRUE;
    }
    if(header_newp2->head==second) {
        header_newp2->head=second->next;
    }
    if((second!=prev2)&&(end!=TRUE)) {
        prev2->next=prev2->next->next;
        second=second->next;
    }
    else {
        prev2=prev2->next;
        second=second->next;
    }
    if(header_newp1->tail==first) {
        header_newp1->tail=prev1;
        prev1->next=NULL;
        stop=TRUE;
    }
    if(header_newp1->head==first) {
        header_newp1->head=first->next;
    }
}

```

```

        if((first!=prev1)&&(done!=TRUE)) {
            prev1->next=prev1->next->next;
            first=first->next;
        }
        else {
            prev1=prev1->next;
            first=first->next;
        }
        header_newp1->length--;
        header_newp2->length--;
        header_newn1->length--;
        header_newn2->length--;
        curr=head_2clinv->tail;
    }
    else {
        prev4=forth;
        forth=forth->next;
    }
    if(header_newn2->length==0) break;
}
if(header_newn1->length==0) break;
if(finished!=TRUE) {
    prev3=third;
    third=third->next;
}
forth=header_newn2->head;
prev4=forth;
}
if(header_newp2->length==0) break;
if(end!=TRUE) {
    prev2=second;
    second=second->next;
}
third=header_newn1->head;
prev3=third;
forth=header_newn2->head;
prev4=forth;
}
if(header_newp1->length==0) break;
if(done!=TRUE) {
    prev1=first;
    first=first->next;
}
second=header_newp2->head;
prev2=second;

```

```

    third=header_newn1->head;
    prev3=third;
    forth=header_newn2->head;
    prev4=forth;
}
}

```

```

/*****
* This function identifies all 2 input NOR gates and places them in a link *
* list.
*****/

```

Function comparenor2()

```

{
    int finished;
    int end;
    int done;
    int complete;
    int stop;

    trans *first, *second, *third, *forth;
    trans *prev1, *prev2, *prev3, *prev4;
    nor2 *curr;
    nor2 *newnor2;

    first=header_newp1->head;
    second=header_newp2->head;
    third=header_newn2->head;
    forth=header_newn2->head;

    prev1=first;
    prev2=second;
    prev3=third;
    prev4=forth;

    totallength=totalwidth=0;
    stop=FALSE;

    while((first !=NULL)&&(stop == FALSE)) {
        complete=FALSE;
        done=FALSE;
        while((second !=NULL)&&(!done)) {
            end=FALSE;
            while((third !=NULL)&&(!end)) {

```

```

finished=FALSE;
while((forth !=NULL)&&(!finished)) {
    if (((strcmp(first->gate,forth->gate))==0)&&
        ((strcmp(second->gate,third->gate))==0)&&
        ((strcmp(second->gate,first->gate))!=0)&&
        ((strcmp(third->drain,forth->drain))==0)&&
        (((strcmp(second->drain,first->drain))==0)&&
        ((strcmp(third->drain,first->source))==0))) ||
        (((strcmp(second->drain,first->source))==0)&&
        ((strcmp(third->drain,first->drain))==0)))) {
        head_nor2->length++;
        curr=head_nor2->tail;
        finished=TRUE;
        newnor2=Newnor2();
        if(head_nor2->head == NULL) {
            head_nor2->head=newnor2;
            head_nor2->tail=newnor2;
        }
        else {
            head_nor2->tail=newnor2;
            curr->next=newnor2;
        }
        newnor2->input1=malloc(sizeof(char)*strlen(second->gate)+1);
        newnor2->input2=malloc(sizeof(char)*strlen(first->gate)+1);
        newnor2->output=malloc(sizeof(char)*strlen(forth->drain)+1);

        strcpy(newnor2->input1,second->gate);
        strcpy(newnor2->input2,first->gate);
        strcpy(newnor2->output,forth->drain);
        if (((strcmp(first->length,second->length))==0)&&
            ((strcmp(first->length,forth->length))==0)&&
            ((strcmp(second->length,third->length))==0)) {
            newnor2->length=malloc(sizeof(char)*strlen(first->length)+1);
            strcpy(newnor2->length,first->length);
        }
        else if (((strcmp(second->length,first->length))==0)&&
            ((strcmp(third->length,forth->length))!=0)) {
            totallength=(strlen(first->length)+strlen(third->length));
            totallength=(strlen(forth->length)+totallength+3);
            newnor2->length=malloc(sizeof(char)*(totallength)+1);

            strcpy(newnor2->length,first->length);
            strcat(newnor2->length,"--");
            strcat(newnor2->length,third->length);
            strcat(newnor2->length,"-");
        }
    }
}

```

```

        strcat(newnor2->length,forth->length);
    }
    else if(((strcmp(first->length,second->length))!=0)&&
        ((strcmp(third->length,forth->length))==0)) {
        totallength=(strlen(first->length)+strlen(second->length));
        totallength=(strlen(third->length)+totallength+3);
        newnor2->length=malloc(sizeof(char)*(totallength)+1);

        strcpy(newnor2->length,first->length);
        strcat(newnor2->length,"-");
        strcat(newnor2->length,second->length);
        strcat(newnor2->length,"--");
        strcat(newnor2->length,third->length);
    }
    else {
        totallength=(strlen(first->length)+strlen(second->length));
        totallength=(strlen(third->length)+strlen(forth->length)
            +totallength+4);
        newnor2->length=malloc(sizeof(char)*(totallength)+1);

        strcpy(newnor2->length,first->length);
        strcat(newnor2->length,"-");
        strcat(newnor2->length,second->length);
        strcat(newnor2->length,"--");
        strcat(newnor2->length,third->length);
        strcat(newnor2->length,"-");
        strcat(newnor2->length,forth->length);
    }
    if(((strcmp(first->width,second->width))==0)&&
        ((strcmp(first->width,forth->width))==0)&&
        ((strcmp(second->width,third->width))==0)) {
        newnor2->width=malloc(sizeof(char)*strlen(first->width)+1);
        strcpy(newnor2->width,first->width);
    }
    else if(((strcmp(second->width,first->width))==0)&&
        ((strcmp(third->width,forth->width))!=0)) {
        totalwidth=(strlen(first->width)+strlen(third->width)+
            strlen(forth->width)+3);
        newnor2->width=malloc(sizeof(char)*(totalwidth)+1);
        strcpy(newnor2->width,first->width);
        strcat(newnor2->width,"--");
        strcat(newnor2->width,third->width);
        strcat(newnor2->width,"-");
        strcat(newnor2->width,forth->width);
    }
}

```

```

else if(((strcmp(first->width,second->width))!=0)&&
        ((strcmp(third->width,forth->width))==0)) {
    totalwidth=(strlen(first->width)+strlen(second->width)+
                strlen(third->width)+3);
    newnor2->width=malloc(sizeof(char)*(totalwidth)+1);

    strcpy(newnor2->width,first->width);
    strcat(newnor2->width,"-");
    strcat(newnor2->width,second->width);
    strcat(newnor2->width,"--");
    strcat(newnor2->width,third->width);
}
else {
    totalwidth=(strlen(second->width)+strlen(third->width)+
                strlen(first->width)+strlen(forth->width)+4);
    newnor2->width=malloc(sizeof(char)*(totalwidth)+1);

    strcpy(newnor2->width,first->width);
    strcat(newnor2->width,"-");
    strcat(newnor2->width,second->width);
    strcat(newnor2->width,"--");
    strcat(newnor2->width,third->width);
    strcat(newnor2->width,"-");
    strcat(newnor2->width,forth->width);
}
if(header_newn2->tail==forth) {
    header_newn2->tail=prev4;
    prev4->next=NULL;
    complete=TRUE;
}
if(header_newn2->head==forth) {
    header_newn2->head=forth->next;
}
if((forth!=prev4)&&(complete!=TRUE)) {
    prev4->next=prev4->next->next;
    forth=forth->next;
}
else {
    prev4=prev4->next;
    forth=forth->next;
}
if(header_newn2->tail==third) {
    header_newn2->tail=prev3;
    prev3->next=NULL;
    end=TRUE;
}

```

```

}
if(header_newn2->head==third) {
    header_newn2->head=third->next;
}
if((third!=prev3)&&(finished!=TRUE)) {
    prev3->next=prev3->next->next;
    third=third->next;
}
else {
    prev3=prev3->next;
    third=third->next;
}
if(header_newp2->tail==second) {
    header_newp2->tail=prev2;
    prev2->next=NULL;
    done=TRUE;
}
if(header_newp2->head==second) {
    header_newp2->head=second->next;
}
if((second!=prev2)&&(end!=TRUE)) {
    prev2->next=prev2->next->next;
    second=second->next;
}
else {
    prev2=prev2->next;
    second=second->next;
}
if(header_newp1->tail==first) {
    header_newp1->tail=prev1;
    prev1->next=NULL;
    stop=TRUE;
}
if(header_newp1->head==first) {
    header_newp1->head=first->next;
}
if((first!=prev1)&&(done!=TRUE)) {
    prev1->next=prev1->next->next;
    first=first->next;
}
else {
    prev1=prev1->next;
    first=first->next;
}
header_newp1->length--;

```

```

        header_newp2->length--;
        header_newn2->length--;
        curr=head_nor2->tail;
    }
    else {
        prev4=forth;
        forth=forth->next;
    }
    if(header_newn2->length==0) break;
}
if(header_newn2->length==0) break;
if(finished!=TRUE) {
    prev3=third;
    third=third->next;
}
forth=header_newn2->head;
prev4=forth;
}
if(header_newp2->length==0) break;
if(end!=TRUE) {
    prev2=second;
    second=second->next;
}
third=header_newn2->head;
prev3=third;
forth=header_newn2->head;
prev4=forth;
}
if(header_newp1->length==0) break;
if(done!=TRUE) {
    prev1=first;
    first=first->next;
}
second=header_newp2->head;
prev2=second;
third=header_newn2->head;
prev3=third;
forth=header_newn2->head;
prev4=forth;
}
}

```

```

/*****
* This function identifies the 2 input NAND gates and places them in a
* link list.
*****/

```

Function comparenand2()

```

{
    int finished;
    int end;
    int done;
    int complete;
    int stop;

    trans *first, *second, *third, *forth;
    trans *prev1, *prev2, *prev3, *prev4;
    nand2 *curr;
    nand2 *newnand2;

    first=header_newn1->head;
    second=header_newn2->head;
    third=header_newp2->head;
    forth=header_newp2->head;

    prev1=first;
    prev2=second;
    prev3=third;
    prev4=forth;

    totallength=totalwidth=0;
    stop=FALSE;
    while((first !=NULL)&&(stop == FALSE)) {
        complete=FALSE;
        done=FALSE;
        while((second !=NULL)&&(!done)) {
            end=FALSE;
            while((third !=NULL)&&(!end)) {
                finished=FALSE;
                while((forth !=NULL)&&(!finished)) {
                    if (((strcmp(third->gate,second->gate))==0)&&
                        ((strcmp(forth->gate,first->gate))==0)&&
                        ((strcmp(forth->gate,third->gate))!=0)&&
                        ((strcmp(third->drain,forth->drain))==0)&&
                        (((strcmp(second->drain,first->drain))==0)&&
                        ((strcmp(forth->drain,first->source))==0))||
                        (((strcmp(second->drain,first->source))==0)&&

```

```

        ((strcmp(first->drain,forth->drain))==0))) {
head_nand2->length++;
curr=head_nand2->tail;
finished=TRUE;
newnand2=Newnand2();
if(head_nand2->head == NULL) {
    head_nand2->head=newnand2;
    head_nand2->tail=newnand2;
}
else {
    head_nand2->tail=newnand2;
    curr->next=newnand2;
}
newnand2->input1=malloc(sizeof(char)*strlen(third->gate)+1);
newnand2->input2=malloc(sizeof(char)*strlen(forth->gate)+1);
newnand2->output=malloc(sizeof(char)*strlen(third->drain)+1);
strcpy(newnand2->input1,third->gate);
strcpy(newnand2->input2,forth->gate);
strcpy(newnand2->output,third->drain);
if(((strcmp(third->length,forth->length))==0)&&
    ((strcmp(third->length,second->length))==0)&&
    ((strcmp(forth->length,first->length))==0)) {
    newnand2->length=malloc(sizeof(char)*strlen(third->length)+1);
    strcpy(newnand2->length,third->length);
}
else if (((strcmp(forth->length,third->length))==0)&&
        ((strcmp(first->length,second->length))!=0)) {
    totallength=(strlen(third->length)+strlen(first->length));
    totallength=(strlen(second->length)+totallength+3);
    newnand2->length=malloc(sizeof(char)*(totallength)+1);
    strcpy(newnand2->length,third->length);
    strcat(newnand2->length,"--");
    strcat(newnand2->length,first->length);
    strcat(newnand2->length,"-");
    strcat(newnand2->length,second->length);
}
else if (((strcmp(first->length,second->length))==0)&&
        ((strcmp(third->length,forth->length))!=0)) {
    totallength=(strlen(first->length)+strlen(forth->length));
    totallength=(strlen(third->length)+totallength+3);
    newnand2->length=malloc(sizeof(char)*(totallength)+1);
    strcpy(newnand2->length,third->length);
    strcat(newnand2->length,"-");
    strcat(newnand2->length,forth->length);
    strcat(newnand2->length,"--");
}

```

```

    strcat(newnand2->length,first->length);
}
else {
    totallength=(strlen(first->length)+strlen(second->length));
    totallength=(strlen(third->length)+strlen(forth->length)
        +totallength+4);
    newnand2->length=malloc(sizeof(char)*(totallength)+1);
    strcat(newnand2->length,third->length);
    strcat(newnand2->length,"-");
    strcat(newnand2->length,forth->length);
    strcat(newnand2->length,"--");
    strcpy(newnand2->length,first->length);
    strcat(newnand2->length,"-");
    strcat(newnand2->length,second->length);
}
if(((strcmp(third->width,forth->width))==0)&&
    ((strcmp(third->width,second->width))==0)&&
    ((strcmp(forth->width,first->width))==0)) {
    newnand2->width=malloc(sizeof(char)*strlen(first->width)+1);
    strcpy(newnand2->width,third->width);
}
else if(((strcmp(second->width,first->width))!=0)&&
    ((strcmp(third->width,forth->width))==0)) {
    totalwidth=(strlen(first->width)+strlen(third->width)+
        strlen(second->width)+3);
    newnand2->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newnand2->width,third->width);
    strcat(newnand2->width,"--");
    strcat(newnand2->width,first->width);
    strcat(newnand2->width,"-");
    strcat(newnand2->width,second->width);
}
else if(((strcmp(first->width,second->width))==0)&&
    ((strcmp(third->width,forth->width))!=0))
{
    totalwidth=(strlen(first->width)+strlen(forth->width)+
        strlen(third->width)+3);
    newnand2->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newnand2->width,third->width);
    strcat(newnand2->width,"-");
    strcat(newnand2->width,forth->width);
    strcat(newnand2->width,"--");
    strcat(newnand2->width,first->width);
}
else {

```

```

        totalwidth=(strlen(second->width)+strlen(third->width)+
                    strlen(first->width)+strlen(forth->width)+4);
        newnand2->width=malloc(sizeof(char)*(totalwidth)+1);
        strcpy(newnand2->width,third->width);
        strcat(newnand2->width,"-");
        strcat(newnand2->width,forth->width);
        strcat(newnand2->width,"--");
        strcat(newnand2->width,first->width);
        strcat(newnand2->width,"-");
        strcat(newnand2->width,second->width);
    }
    if(header_newp2->tail==forth) {
        header_newp2->tail=prev4;
        prev4->next=NULL;
        complete=TRUE;
    }
    if(header_newp2->head==forth) {
        header_newp2->head=forth->next;
    }
    if((forth!=prev4)&&(complete!=TRUE)) {
        prev4->next=prev4->next->next;
        forth=forth->next;
    }
    else {
        prev4=prev4->next;
        forth=forth->next;
    }
    if(header_newp2->tail==third) {
        header_newp2->tail=prev3;
        prev3->next=NULL;
        end=TRUE;
    }
    if(header_newp2->head==third) {
        header_newp2->head=third->next;
    }
    if((third!=prev3)&&(finished!=TRUE)) {
        prev3->next=prev3->next->next;
        third=third->next;
    }
    else {
        prev3=prev3->next;
        third=third->next;
    }
    if(header_newn2->tail==second) {
        header_newn2->tail=prev2;
    }

```

```

    prev2->next=NULL;
    done=TRUE;
}
if(header_newn2->head==second) {
    header_newn2->head=second->next;
}
if((second!=prev2)&&(end!=TRUE)) {
    prev2->next=prev2->next->next;
    second=second->next;
}
else {
    prev2=prev2->next;
    second=second->next;
}
if(header_newn1->tail==first) {
    header_newn1->tail=prev1;
    prev1->next=NULL;
    stop=TRUE;
}
if(header_newn1->head==first) {
    header_newn1->head=first->next;
}
if((first!=prev1)&&(done!=TRUE)) {
    prev1->next=prev1->next->next;
    first=first->next;
}
else {
    prev1=prev1->next;
    first=first->next;
}
header_newp2->length--;
header_newn1->length--;
header_newn2->length--;
curr=head_nand2->tail;
}
else {
    prev4=forth;
    forth=forth->next;
}
if(header_newp2->length==0) break;
}
if(header_newp2->length==0) break;
if(finished!=TRUE) {
    prev3=third;
    third=third->next;
}

```

```

    }
    forth=header_newp2->head;
    prev4=forth;
}
if(header_newn2->length==0) break;
if(end!=TRUE) {
    prev2=second;
    second=second->next;
}
hird=header_newp2->head;
prev3=third;
forth=header_newp2->head;
prev4=forth;
}
f(header_newn1->length==0) break;
if(done!=TRUE) {
    prev1=first;
    f first=first->next;
}
second=header_newn2->head;
prev2=second;
third=header_newp2->head;
3=third;
h=header_newp2->head;
forth;
}
}

```

```

/*****
* This function identifies the precharged 2 input NOR and NAND gates and *
* places them in a separate link list. The searching is done concurrently *
* in order to achieve higher execution speed. *
*****/

```

```

Function compareprecharged()
{
    int finished;
    int end;
    int done;
    int complete;
    int stop;

    trans *first, *second, *third, *forth;
    trans *prev1, *prev2, *prev3, *prev4;

```

```

prenand2 *curr;
prenand2 *newprenand2;
prenor2 *curr1;
prenor2 *newprenor2;

first=header_newp2->head;
second=header_newn1->head;
third=header_newn1->head;
forth=header_newn2->head;

prev1=first;
prev2=second;
prev3=third;
prev4=forth;

totallength=totalwidth=0;
stop=FALSE;

while((first !=NULL)&&(stop == FALSE)) {
    complete=FALSE;
    done=FALSE;

    while((second !=NULL)&&(!done)) {
        end=FALSE;

        while((third !=NULL)&&(!end)) {
            finished=FALSE;

            while((forth !=NULL)&&(!finished)) {
                if (((strcmp(first->gate,forth->gate))==0)&&
                    ((strcmp(second->gate,third->gate))!=0)&&
                    ((strcmp(second->gate,first->gate))!=0)&&
                    ((strcmp(first->gate,third->gate))!=0)&&
                    (((strcmp(first->drain,second->drain))==0)&&
                    ((strcmp(third->drain,forth->drain))==0)&&
                    ((strcmp(second->source,third->source))==0))) ||
                    (((strcmp(first->drain,second->drain))==0)&&
                    ((strcmp(second->source,third->drain))==0)&&
                    ((strcmp(third->source,forth->drain))==0))) ||
                    (((strcmp(first->drain,second->source))==0)&&
                    ((strcmp(second->drain,third->source))==0)&&
                    ((strcmp(third->drain,forth->drain))==0))) ||
                    (((strcmp(first->drain,second->source))==0)&&
                    ((strcmp(third->drain,second->drain))==0)&&

```

```

        ((strcmp(third->source,forth->drain))==0)))) {
head_prenand2->length++;
curr=head_prenand2->tail;
finished=TRUE;
newprenand2=Newprenand2();

if(head_prenand2->head == NULL) {
    head_prenand2->head=newprenand2;
    head_prenand2->tail=newprenand2;
}
else {
    head_prenand2->tail=newprenand2;
    curr->next=newprenand2;
}

newprenand2->input1=malloc(sizeof(char)*strlen(second->gate)+1);
newprenand2->input2=malloc(sizeof(char)*strlen(third->gate)+1);
newprenand2->phase=malloc(sizeof(char)*strlen(first->gate)+1);
newprenand2->output=malloc(sizeof(char)*strlen(first->drain)+1);
strcpy(newprenand2->input1,second->gate);
strcpy(newprenand2->input2,third->gate);
strcpy(newprenand2->phase,first->gate);
strcpy(newprenand2->output,first->drain);
if(((strcmp(second->length,third->length))==0)&&
    ((strcmp(second->length,forth->length))==0)) {
    totallength=(strlen(first->length)+strlen(second->length)+2);
    newprenand2->length=malloc(sizeof(char)*(totallength)+1);
    strcpy(newprenand2->length,first->length);
    strcat(newprenand2->length,"--");
    strcat(newprenand2->length,second->length);
}
else {
    totallength=(strlen(first->length)+strlen(second->length)+
        strlen(third->length)+strlen(forth->length)+3);
    newprenand2->length=malloc(sizeof(char)*(totallength)+1);
    strcpy(newprenand2->length,first->length);
    strcat(newprenand2->length,"--");
    strcat(newprenand2->length,second->length);
    strcat(newprenand2->length,"-");
    strcat(newprenand2->length,third->length);
    strcat(newprenand2->length,"-");
    strcat(newprenand2->length,forth->length);
}

if(((strcmp(second->width,third->width))==0)&&
    ((strcmp(second->width,forth->width))==0)) {
    totalwidth=(strlen(first->width)+strlen(second->width)+2);

```

```

    newprenand2->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newprenand2->width,first->width);
    strcat(newprenand2->width,"--");
    strcat(newprenand2->width,second->width);
}
else {
    totalwidth=(strlen(first->width)+strlen(second->width)+
                strlen(third->width)+strlen(forth->width)+4);
    newprenand2->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newprenand2->width,first->width);
    strcat(newprenand2->width,"--");
    strcat(newprenand2->width,second->width);
    strcat(newprenand2->width,"-");
    strcat(newprenand2->width,third->width);
    strcat(newprenand2->width,"-");
    strcat(newprenand2->width,forth->width);
}
if(header_newn2->tail==forth) {
    header_newn2->tail=prev4;
    prev4->next=NULL;
    complete=TRUE;
}
if(header_newn2->head==forth) {
    header_newn2->head=forth->next;
}
if((forth!=prev4)&&(complete!=TRUE)) {
    prev4->next=prev4->next->next;
    forth=forth->next;
}
else {
    prev4=prev4->next;
    forth=forth->next;
}
if(header_newn1->tail==third) {
    header_newn1->tail=prev3;
    prev3->next=NULL;
    end=TRUE;
}
if(header_newn1->head==third) {
    header_newn1->head=third->next;
}
if((third!=prev3)&&(finished!=TRUE)) {
    prev3->next=prev3->next->next;
    third=third->next;
}
}

```

```

else {
    prev3=prev3->next;
    third=third->next;
}
if(header_newn1->tail==second) {
    header_newn1->tail=prev2;
    prev2->next=NULL;
    done=TRUE;
}
if(header_newn1->head==second) {
    header_newn1->head=second->next;
}
if((second!=prev2)&&(end!=TRUE)) {
    prev2->next=prev2->next->next;
    second=second->next;
}
else {
    prev2=prev2->next;
    second=second->next;
}
if(header_newp2->tail==first) {
    header_newp2->tail=prev1;
    prev1->next=NULL;
    stop=TRUE;
}
if(header_newp2->head==first) {
    header_newp2->head=first->next;
}
if((first!=prev1)&&(done!=TRUE)) {
    prev1->next=prev1->next->next;
    first=first->next;
}
else {
    prev1=prev1->next;
    first=first->next;
}
header_newp2->length--;
header_newn1->length--;
header_newn2->length--;
curr=head_prenand2->tail; /* end of nand2*/
}

else if (((strcmp(first->gate,forth->gate))==0)&&
        ((strcmp(second->gate,third->gate))!=0)&&

```

```

        ((strcmp(second->gate,first->gate))!=0)&&
        ((strcmp(first->gate,third->gate))!=0)&&
        (((strcmp(first->drain,second->drain))==0)&&
        ((strcmp(third->drain,forth->drain))==0)&&
        ((strcmp(third->source,second->drain))==0)&&
        ((strcmp(second->source,third->drain))==0))||
        (((strcmp(first->drain,second->drain))==0)&&
        ((strcmp(second->source,third->source))==0)&&
        ((strcmp(forth->drain,third->source))==0)&&
        ((strcmp(third->drain,second->drain))==0))||
        (((strcmp(first->drain,second->source))==0)&&
        ((strcmp(second->drain,third->source))==0)&&
        ((strcmp(third->drain,second->source))==0)&&
        ((strcmp(third->source,forth->drain))==0))||
        (((strcmp(first->drain,second->source))==0)&&
        ((strcmp(third->drain,second->drain))==0)&&
        ((strcmp(third->source,second->source))==0)&&
        ((strcmp(third->drain,forth->drain))==0)))) {
head_prenor2->length++;
curri=head_prenor2->tail;
finished=TRUE;
newprenor2=Newprenor2();

if(head_prenor2->head == NULL) {
    head_prenor2->head=newprenor2;
    head_prenor2->tail=newprenor2;
}
else {
    head_prenor2->tail=newprenor2;
    curri->next=newprenor2;
}
newprenor2->input1=malloc(sizeof(char)*strlen(second->gate)+1);
newprenor2->input2=malloc(sizeof(char)*strlen(third->gate)+1);
newprenor2->phase=malloc(sizeof(char)*strlen(first->gate)+1);
newprenor2->output=malloc(sizeof(char)*strlen(first->drain)+1);
strcpy(newprenor2->input1,second->gate);
strcpy(newprenor2->input2,third->gate);
strcpy(newprenor2->phase,first->gate);
strcpy(newprenor2->output,first->drain);
if(((strcmp(second->length,third->length))==0)&&
    ((strcmp(second->length,forth->length))==0)) {
    totallength=(strlen(first->length)+strlen(second->length)+2);
    newprenor2->length=malloc(sizeof(char)*(totallength)+1);
    strcpy(newprenor2->length,first->length);
    strcat(newprenor2->length,"--");
}

```

```

    strcat(newprenor2->length,second->length);
}
else {
    totallength=(strlen(first->length)+strlen(second->length)+
        strlen(third->length)+strlen(forth->length)+4);
    newprenor2->length=malloc(sizeof(char)*(totallength)+1);
    strcpy(newprenor2->length,first->length);
    strcat(newprenor2->length,"--");
    strcat(newprenor2->length,second->length);
    strcat(newprenor2->length,"-");
    strcat(newprenor2->length,third->length);
    strcat(newprenor2->length,"-");
    strcat(newprenor2->length,forth->length);
}
if(((strcmp(second->width,third->width))==0)&&
    ((strcmp(second->width,forth->width))==0)) {
    totalwidth=(strlen(first->width)+strlen(second->width)+2);
    newprenor2->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newprenor2->width,first->width);
    strcat(newprenor2->width,"--");
    strcat(newprenor2->width,second->width);
}
else {
    totalwidth=(strlen(first->width)+strlen(second->width)+
        strlen(third->width)+strlen(forth->width)+4);
    newprenor2->width=malloc(sizeof(char)*(totalwidth)+1);
    strcpy(newprenor2->width,first->width);
    strcat(newprenor2->width,"--");
    strcat(newprenor2->width,second->width);
    strcat(newprenor2->width,"-");
    strcat(newprenor2->width,third->width);
    strcat(newprenor2->width,"-");
    strcat(newprenor2->width,forth->width);
}
if(header_newn2->tail==forth) {
    header_newn2->tail=prev4;
    prev4->next=NULL;
    complete=TRUE;
}
if(header_newn2->head==forth) {
    header_newn2->head=forth->next;
}
if((forth!=prev4)&&(complete!=TRUE)) {
    prev4->next=prev4->next->next;
    forth=forth->next;
}

```

```

}
else {
    prev4=prev4->next;
    forth=forth->next;
}
if(header_newn1->tail==third) {
    header_newn1->tail=prev3;
    prev3->next=NULL;
    end=TRUE;
}
if(header_newn1->head==third) {
    header_newn1->head=third->next;
}
if((third!=prev3)&&(finished!=TRUE)) {
    prev3->next=prev3->next->next;
    third=third->next;
}
else {
    prev3=prev3->next;
    third=third->next;
}
if(header_newn1->tail==second) {
    header_newn1->tail=prev2;
    prev2->next=NULL;
    done=TRUE;
}
if(header_newn1->head==second) {
    header_newn1->head=second->next;
}
if((second!=prev2)&&(end!=TRUE)) {
    prev2->next=prev2->next->next;
    second=second->next;
}
else {
    prev2=prev2->next;
    second=second->next;
}
if(header_newp2->tail==first) {
    header_newp2->tail=prev1;
    prev1->next=NULL;
    top=TRUE;
}
if(header_newp2->head==first) {
    header_newp2->head=first->next;
}

```

```

        if((first!=prev1)&&(done!=TRUE)) {
            prev1->next=prev1->next->next;
            first=first->next;
        }
        else {
            prev1=prev1->next;
            first=first->next;
        }
        header_newp2->length--;
        header_newn1->length--;
        header_newn2->length--;
        curr1=head_prenor2->tail;
    }
    else {
        prev4=forth;
        forth=forth->next;
    }
    if(header_newn2->length==0) break;
}
if(header_newn1->length==0) break;
if(finished!=TRUE) {
    prev3=third;
    third=third->next;
}
forth=header_newn2->head;
prev4=forth;
}
if(header_newn1->length==0) break;
if(end!=TRUE) {
    prev2=second;
    second=second->next;
}
third=header_newn1->head;
prev3=third;
forth=header_newn2->head;
prev4=forth;
}
if(header_newp2->length==0) break;
if(done!=TRUE) {
    prev1=first;
    first=first->next;
}
second=header_newp2->head;
prev2=second;
third=header_newn1->head;

```

```
    prev3=third;  
    forth=header_newn2->head;  
    prev4=forth;  
  }  
}
```

C. STRUCTURE RECOGNITION

```

/*****
 * This function performs the recognition of the big file's abstract
 * structures.
 *****/

Function comparestructures1()
{
    trans *first, *prev1st, *second, *prev2nd;
    char c;
    int nfound, pvdd;
    int i;

    first=header1_newp->head;
    prev1st=first;
    nump1= numn1=0;
    numdevice1= numtrans1=1;
    ground=FALSE;
    nfound=FALSE;
    pvdd=FALSE;

    while(first!=NULL) {
        if(nfound==TRUE) { /* 2nd time */
            nfound=FALSE;
            checkp1(nump1,numtrans1-1);
        }
        else if(((strcmp(first->gate,"Vdd")==0)||
                ((strcmp(first->drain,"Vdd")==0)||
                ((strcmp(first->source,"Vdd")==0)))
        {
            stacknum1=1;
            Push1(first);
            combinep1(first,prev1st);
            pvdd=TRUE;
            nump1=numtrans1-1;
            nfound=FALSE;

            for (i=1; i<=nump1; i++)
            {
                /* Find all of the n-type transistors that
                connect to the p-type
                transistors already found. */

                if(nfound==TRUE) break;
            }
        }
    }
}

```

```

second=header1_newn->head;
if(second==NULL)break; /* no more N type; exit */
prev2nd=second;
/*****
** Does the n-type transistor connect to the p-type transistor?      **
*****/
while((second!=NULL)&&(nfound==FALSE))
{
    if(((strcmp(second->source,structure1[numdevice1][i]->source))==0)||
        ((strcmp(second->source,structure1[numdevice1][i]->drain))==0)||
        ((strcmp(second->drain,structure1[numdevice1][i]->source))==0)||
        ((strcmp(second->drain,structure1[numdevice1][i]->drain))==0))
    {
stacknum1=1;
Push1(second);
    combinen1(second,prev2nd);
nfound=TRUE;
    }
else {
    prev2nd=second;
second=second->next;
    }
}
}
/*****
** Has a p-type transistor which connects to Vdd been found?      **
** Yes, has a matching n-type been found? No, error in .sim      **
** file because a Vdd transistor must go to an n-type sometime. **
*****/
if((pvdd==TRUE)&&(ground==FALSE)) {
tcount1[numdevice1]=numtrans1-1;
    print_error();
    printf("Do you want to quit and check your .sim file? (y or n)");
    c=getchar();
    while((c!='y')&&(c!='n')) {
        printf("(y or n)");
        c=getchar();
    }
    if(c=='y') {
print_structures1();
exit(0);
    }
    else ground=TRUE;
}

```

```

    }
    /*****
    ** Has a p-type transistor which connects to Vdd been found?    **
    ** No, continue the search through the linked list.              **
    *****/
    else if(pvdd==FALSE) {
    prev1st=first;
    first=first->next;
    }
    /*****
    ** A partial level1 device has been found. Repeat the loop and **
    ** see if there is another transistor in this device.          **
    *****/
    else if(nfound==TRUE) {
    first=header1_newp->head;
    prev1st=first;
    }
    /*****
    ** A level1 device has been found reset the search pointers and **
    ** see if there is another level1 device in the circuit.        **
    *****/
    else {
    tcount1[numdevice1]=numtrans1-1;
    numdevice1++;
    numtrans1=1;
    ground=FALSE;
    nfound=FALSE;
    pvdd=FALSE;
    if(header1_newp->length==0) break;
    else
    first=header1_newp->head;
    prev1st=first;
    }
    }
    /*****
    ** Determine whether above section was exited before numdevice **
    ** was incremented.
    *****/
    if(nfound==TRUE) {
    tcount1[numdevice1]=numtrans1-1;
    nfound=FALSE;
    numdevice1++;
    }
    /*****
    ** There are no transistors connected to Vdd left connect      **

```

```

** the remaining transistors in the file.                                     **
*****/
{
/*****
** Combine the two transistor lists for easy recursive compares.**
*****/
header1_new = create();
if(header1_newp->head!=NULL) {
    header1_new->head=header1_newp->head;
    header1_newp->tail->next=header1_newn->head;
}
else {
    header1_new->head=header1_newn->head;
}
header1_new->tail=header1_newn->tail;
first = header1_new->head;
/*****
** There are no transistors connected to Vdd left connect                **
** the remaining transistors in the file.                                **
*****/
while(first!=NULL)
{
    stacknum1=1;
    numtrans1=1;
    prev1st=first;
    Push1(first);
    combine1(first,prev1st);
    first = header1_new->head;
    tcount1[numdevice1]=numtrans1-1;
    numdevice1++;
}
}
/*****
** Prepare numdevice counter for printing.                                **
*****/
header1_newp->length=0;
header1_newn->length=0;
numdevice1=numdevice1-1;
}

/*****
* This function pushes a transistor in the stack.                          *
*****/

```

```

Function Push1(T)
trans  *T;
{
stack1[stacknum1] = T;
stacknum1++;
if((strcmp(T->type,"p")==0) {
    if((strcmp(T->source,"Vdd")==0) {
        p2count1[numdevice1]++;
        pcount1[numdevice1]++;
    }
    else{
        p1count1[numdevice1]++;
        pcount1[numdevice1]++;
    }
}
else {
    if((strcmp(T->source,"GND")==0) {
        n2count1[numdevice1]++;
        ncount1[numdevice1]++;
    }
    else {
        ncount1[numdevice1]++;
        n1count1[numdevice1]++;
    }
}
}

/*****
** This function removes the transistor from the stack and
** places it into the "structure1" array.
*****/

Function Pop1()
{
trans *tt;
tt = stack1[stacknum1-1];
structure1[numdevice1][numtrans1++] = stack1[--stacknum1];
}

/*****
** This function compares the transistor with all of the
** in the list to find a match. It calls itself recursively
** until all matches are found.
*****/

```

```

*****/

Function combine1(start,previst)
trans *start, *previst;
{
    trans *compare;
    int set;
    set=FALSE;
    /*****
    ** Remove the transistor that now is part of an abstract structure. **
    ** If the transistor lists' head or tail pointer are to be **
    ** deleted change the head or tail. Delete the used transistor **
    ** and decrease the transistor lists' length. **
    *****/
    compare = start;
    if(header1_new->length == 1) {
        set = TRUE;
        header1_new->length = 0;
        header1_new->head = NULL;
        start->next=NULL;
    }
    else { if(start== header1_new->head &&set!=TRUE) {
        header1_new->head = start->next;
        header1_new->length--;
    }
    else { if(start== header1_new->tail &&set!=TRUE) {
        header1_new->tail = previst;
        previst->next=NULL;
        header1_new->length--;
    }
    else {
        previst->next=previst->next->next;
        start->next=NULL;
        header1_new->length--;
    }
    }
}
/*****
** Go to the head of the list to begin the comparisons. **
*****/
if(header1_new->head != NULL)
{
    start = header1_new->head;
}
else {

```

```

        start = NULL;
    }
    while(start != NULL)
    {
        /*****
        ** Are there any transistors that can be connected into a level1**
        ** device.                                                    **
        *****/
        if(((strcmp(start->source,compare->source))==0)||
            ((strcmp(start->source,compare->drain))==0)||
            ((strcmp(start->drain,compare->source))==0)||
            ((strcmp(start->drain,compare->drain))==0))
        {
            /*****
            ** A match was found, place the transistor on the stack and    **
            ** find any transistors which connect to it.                  **
            *****/
            Push1(start);
            combine1(start,prev1st);
            /*****
            ** The recursive call comes back to here. Reset the searching  **
            ** pointers to continue the search.                            **
            *****/
            start=header1_new->head;
        }
        else
        {
            /*****
            ** No match was found increment the pointers.                  **
            *****/
            prev1st=start;
            start=start->next;
        }
    }
    /*****
    ** No more transistors to compare this time. Place the                **
    ** into the structure1 device array.                                  **
    *****/
    Pop1();
}

/*****
** This function compares the transistor with all of the                **
** in the list to find a match. It calls itself recursively             **
*****/

```

```

** until all matches are found.                                     **
*****/

Function combinep1( startp,previst)
trans *startp, *previst;

{
    trans *compare;
    int    set;
    set=FALSE;
/*****
** Remove the transistor that now is part of a level 1 device.  **
** If the transistor lists' head or tail pointer are to be      **
** deleted change the head or tail. Delete the used transistor  **
** and decrease the transistor lists' length.                    **
*****/
    compare = startp;
    if(header1_newp->length == 1) {
        set = TRUE;
        header1_newp->length = 0;
        header1_newp->head = NULL;
        startp->next=NULL;
    }
    else {
        if(startp== header1_newp->head &&set!=TRUE) {
            header1_newp->head = startp->next;
            header1_newp->length--; }
        else {
            if(startp== header1_newp->tail &&set!=TRUE) {
                header1_newp->tail = previst;
                previst->next=NULL;
                header1_newp->length--;
            }
            else {
                previst->next=previst->next->next;
                startp->next=NULL;
                header1_newp->length--;
            }
        }
    }
/*****
** Go to the head of the list to begin the comparisons.        **
*****/
if(header1_newp->head != NULL)
{

```

```

    startp = header1_newp->head;
    prev1st=startp;
}
else {
    startp = NULL;
}
while(startp != NULL)
{
/*****
** Are there any transistors that can be connected into a      **
** structure.                                                    **
*****/
    if (((strcmp(startp->source,compare->source))==0)&&
        ((strcmp(startp->source,"Vdd"))!=0))||
        (((strcmp(startp->source,compare->drain))==0)&&
        ((strcmp(startp->source,"Vdd"))!=0))||
        (((strcmp(startp->drain,compare->source))==0)&&
        ((strcmp(startp->source,"Vdd"))!=0))||
        (((strcmp(startp->drain,compare->drain))==0)&&
        ((strcmp(startp->source,"Vdd"))!=0)))
    {
/*****
** A match was found, place the transistor on the stack and      **
** find any transistors which connect to it.                    **
*****/
        Push1(startp);
        combine1(startp,prev1st);
/*****
** The recursive call comes back to here. Reset the searching    **
** pointers to continue the search.                              **
*****/
        startp=header1_newp->head;
    }
    else
    {
/*****
** No match was found increment the pointers.                    **
*****/
        prev1st=startp;
        startp=startp->next;
    }
}
/*****
** No more transistors to compare this time. Place the          **
** into the level1 device array.                                 **
*****/

```

```

*****/
Pop1();
}

/*****
** This function compares the transistor with all of the **
** in the list to find a match. It calls itself recursively **
** until all matches are found. **
*****/

Function combinen1(startn,previst)
trans *startn, *previst;

{
trans *compare;
int set;
compare = startn;
set = FALSE;
/*****
** Does this n-type transistor connect to ground? **
*****/
    if((strcmp("GND",startn->source))==0)
    {
        ground=TRUE;
    }
/*****
** Remove the transistor that now is part of a structure. **
** If the transistor lists' head or tail pointer are to be **
** deleted change the head or tail. Delete the used transistor **
** and decrease the transistor lists' length. **
*****/
if(header1_newn->length == 1)
{
    set = TRUE;
    header1_newn->length = 0;
    header1_newn->head = NULL;
    startn->next=NULL;
}
else { if(startn== header1_newn->head &&set!=TRUE)
{
    header1_newn->head = startn->next;
    header1_newn->length--;
}
}
else { if(startn== header1_newn->tail &&set!=TRUE)

```

```

{
    header1_newn->tail = prev1st;
    prev1st->next=NULL;
    header1_newn->length--;
}
else
{
    prev1st->next=prev1st->next->next;
    startn->next=NULL;
    header1_newn->length--;
}}}
/*****
** Go to the head of the list to begin the comparisons.      **
*****/
if(header1_newn->head != NULL)
{
    startn = header1_newn->head;
    prev1st = startn;
}
else{
    startn = NULL;
}
while(startn != NULL)
{
/*****
** Are there any transistors that can be connected to this    **
** device.                                                       **
*****/
    if((((strcmp(startn->source,compare->source))==0)&&
        ((strcmp(startn->source,"GND"))!=0))||
        (((strcmp(startn->source,compare->drain))==0)&&
        ((strcmp(startn->source,"GND"))!=0))||
        ((strcmp(startn->drain,compare->source))==0)||
        ((strcmp(startn->drain,compare->drain))==0))
        {
/*****
** A match was found, place the transistor on the stack and    **
** find any transistors which connect to it.                   **
*****/
            Push1(startn);
            combinen1(startn,prev1st);
/*****
** The recursive call comes back to here. Reset the searching  **
** pointers to continue the search.                             **
*****/

```

```

        startn=header1_newn->head;
    }
    else
    {
        /*****
        ** No match was found increment the pointers.          **
        *****/
        prev1st=startn;
        startn=startn->next;
    }
}
/*****
** No more transistors to compare this time. Place the      **
** into the level1 device array.                            **
*****/
Pop1();
}

/*****
** Function checkp(lo,hi)                                     **
** This function compares the transistor with all of the    **
** in the list to find a match. It calls itself recursively **
** until all matches are found.                             **
*****/

Function checkp1(lo,hi)
int lo,hi;

{
    int pfound,nfound,i,j;
    int hi2nd,lasthi;
    trans *first, *prev1st, *second, *prev2nd;
    nfound=FALSE;
    first=header1_newp->head;
    prev1st=first;
    second=header1_newn->head;
    prev2nd=second;

    while(first!=NULL) {
        pfound=FALSE;
        for(i=lo;i<=hi;i++) {
            if((((strcmp(first->source,structure1[numdevice1][i]->source))==0)&&
                ((strcmp(first->source,"Vdd")!=0))||
                (((strcmp(first->source,structure1[numdevice1][i]->drain))==0)&&

```

```

        ((strcmp(first->source,"Vdd")!=0))||
        (((strcmp(first->drain,structure1[numdevice1][i]->source))==0)&&
        ((strcmp(first->source,"Vdd")!=0))||
        (((strcmp(first->drain,structure1[numdevice1][i]->drain))==0)&&
        ((strcmp(first->source,"Vdd")!=0)))
    {
        pfound=TRUE;
        Push1(first);
        combinep1(first,prev1st);
        2nd=numtrans1-1;
        for(j=hi;j<=hi2nd;j++) {
            while(second!=NULL) {
                if(((strcmp(second->source,structure1[numdevice1][i]->source))==0)||
                ((strcmp(second->source,structure1[numdevice1][i]->drain))==0)||
                ((strcmp(second->drain,structure1[numdevice1][i]->source))==0)||
                ((strcmp(second->drain,structure1[numdevice1][i]->drain))==0))
                {
stacknum1=1;
Push1(second);
        combinep1(second,prev2nd);
nfound=TRUE;
second=header1_newn->head;
prev2nd=second;
        lasthi=numtrans1-1;
        }
        else {
            prev2nd=second;
second=second->next;
        }
        }
    }
    if(pfound==TRUE) {
        first=header1_newp->head;
        prev1st=first;
        break;
    }
}
if((pfound!=FALSE)&&(first!=NULL)) {
    prev1st=first;
    first=first->next;
}
}
if(nfound==TRUE) {
    checkp1(hi2nd,lasthi);
}

```

```

    }
}

```

```

/*****
* This function recognizes the small file's abstract structures. *
*****/

```

```

Function comparestructures()

```

```

{
trans    *first, *prev1st, *second, *prev2nd;
char  c;
int  nfound, pvdd;
int  i;

```

```

first=header_newp->head;
prev1st=first;
nump= numn=0;
numdevice= numtrans=1;
ground=FALSE;
nfound=FALSE;
pvdd=FALSE;

```

```

while(first!=NULL) {
    if(nfound==TRUE) { /* 2nd time */
nfound=FALSE;
checkp(nump,numtrans-1);
}
else if(((strcmp(first->gate,"Vdd")==0)||
((strcmp(first->drain,"Vdd")==0)||
((strcmp(first->source,"Vdd")==0))
{
    stacknum=1;
    Push(first);
    combinep(first,prev1st);
    pvdd=TRUE;
    nump=numtrans-1;
    nfound=FALSE;

```

```

for (i=1; i<=nump; i++)
{

```

```

/* Find all of the n-type transistors that
connect to the p-type

```

```

                                transistors already found.    */

    if(nfound==TRUE)    break;
    second=header_newn->head;
    if(second==NULL)break; /* no more N type; exit */
    prev2nd=second;
/*****
** Does the n-type transistor connect to the p-type transistor? **
*****/
    while((second!=NULL)&&(nfound==FALSE))
    {

        if(((strcmp(second->source,structure[numdevice][i]->source))==0)||
            ((strcmp(second->source,structure[numdevice][i]->drain))==0)||
            ((strcmp(second->drain,structure[numdevice][i]->source))==0)||
            ((strcmp(second->drain,structure[numdevice][i]->drain))==0))
        {
            stacknum=1;
            Push(second);
            combinen(second,prev2nd);
            nfound=TRUE;
        }
        else
        {
            prev2nd=second;
            second=second->next;
        }
    }
}

/*****
** Has a p-type transistor which connects to Vdd been found?    **
** Yes, has a matching n-type been found? No, error in .sim    **
** file because a Vdd transistor must go to an n-type sometime. **
*****/
    if((pvdd==TRUE)&&(ground==FALSE)) {
tcount[numdevice]=numtrans-1;
        print_error();
        printf("Do you want to quit and check your .sim file? (y or n)");
        c=getchar();
        while((c!='y')&&(c!='n')) {
            printf("(y or n)");
            c=getchar();
        }
        if(c=='y') {
            print_structures();

```

```

        exit(0);
    }
    else ground=TRUE;
}
/*****
** Has a p-type transistor which connects to Vdd been found?    **
** No, continue the search through the linked list.              **
*****/
    else if(pvdd==FALSE) {
prev1st=first;
first=first->next;
    }
/*****
** A partial level1 device has been found. Repeat the loop and **
** see if there is another transistor in this device.          **
*****/
    else if(nfound==TRUE) {
first=header_newp->head;
prev1st=first;
    }
/*****
** A level1 device has been found reset the search pointers and **
** see if there is another level1 device in the circuit.        **
*****/
    else {
tcount[numdevice]=numtrans-1;
numdevice++;
numtrans1=1;
ground=FALSE;
nfound=FALSE;
pvdd=FALSE;
if(header_newp->length==0) break;
else
first=header_newp->head;
prev1st=first;
    }
}
/*****
** Determine whether above section was exited before numdevice **
** was incremented.                                              **
*****/
if(nfound==TRUE) {
    tcount[numdevice]=numtrans-1;
    nfound=FALSE;
    numdevice++;
}

```

```

}
/*****
** There are no transistors connected to Vdd left connect      **
** the remaining transistors in the file.                      **
*****/
{
/*****
** Combine the two transistor lists for easy recursive compares.**
*****/
header_new = create();
if(header_newp->head!=NULL) {
    header_new->head=header_newp->head;
    header_newp->tail->next=header_newn->head;
}
else {
    header_new->head=header_newn->head;
}
header_new->tail=header_newn->tail;
first = header_new->head;
/*****
** There are no transistors connected to Vdd left connect      **
** the remaining transistors in the file.                      **
*****/
while(first!=NULL)
{
    stacknum=1;
    numtrans=1;
    prev1st=first;
    Push(first);
    combine(first,prev1st);
    first = header_new->head;
    tcount[numdevice]=numtrans-1;
    numdevice++;
}
}
/*****
** Prepare numdevice counter for printing.                    **
*****/
header_newp->length=0;
header_newn->length=0;
numdevice=numdevice-1;
}

/*****

```

```

* This function pushes a transistor into the stack.
*****

```

```

Function Push(T)

```

```

trans *T;

```

```

{
stack[stacknum] = T;
stacknum++;
if((strcmp(T->type,"p"))==0) {
    if((strcmp(T->source,"Vdd"))==0) {
        p2count[numdevice]++;
        pcount[numdevice]++;
    }
    else{
        p1count[numdevice]++;
        pcount[numdevice]++;
    }
}
else {
    if((strcmp(T->source,"GND"))==0) {
        n2count[numdevice]++;
        ncount[numdevice]++;
    }
    else {
        ncount[numdevice]++;
        n1count[numdevice]++;
    }
}
}
}

```

```

/*****
** This function removes the transistor from the stack and
** places it into the level1 (structure1) array.
**
*****/

```

```

Function Pop()

```

```

{
trans *tt;
tt = stack[stacknum-1];
structure[numdevice][numtrans++] = stack[--stacknum];
}

```

```

/*****
** This function compares the transistor with all of the      **
** in the list to find a match. It calls itself recursively  **
** until all matches are found.                               **
*****/

```

```

Function combine(start,previst)

```

```

trans *start, *previst;

```

```

{
    trans *compare;
    int    set;
    set=FALSE;
/*****
** Remove the transistor that now is part of a structure.      **
** If the transistor lists' head or tail pointer are to be    **
** deleted change the head or tail. Delete the used transistor **
** and decrease the transistor lists' length.                  **
*****/
compare = start;
if(header_new->length == 1)
{
    set = TRUE;
    header_new->length = 0;
    header_new->head = NULL;
    start->next=NULL;
}
else {
    if(start== header_new->head &&set!=TRUE) {
        header_new->head = start->next;
        header_new->length--;
    }
    else {
        if(start== header_new->tail &&set!=TRUE) {
            header_new->tail = previst;
            previst->next=NULL;
            header_new->length--;
        }
        else {
            previst->next=previst->next->next;
            start->next=NULL;
            header_new->length--;
        }
    }
}
}

```

```

/*****
** Go to the head of the list to begin the comparisons.      **
*****/
if(header_new->head != NULL) {
    start = header_new->head;
}
else {
    start = NULL;
}
while(start != NULL)
{
/*****
**   Are there any transistors that can be connected into a   **
**   structure?                                              **
*****/
    if(((strcmp(start->source,compare->source))==0)||
        ((strcmp(start->source,compare->drain))==0)||
        ((strcmp(start->drain,compare->source))==0)||
        ((strcmp(start->drain,compare->drain))==0))
        {
/*****
** A match was found, place the transistor on the stack and    **
** find any transistors which connect to it.                  **
*****/
            Push(start);
            combine(start,previst);
/*****
** The recursive call comes back to here. Reset the searching  **
** pointers to continue the search.                            **
*****/
            start=header_new->head;
        }
    else
    {
/*****
** No match was found increment the pointers.                  **
*****/
        previst=start;
        start=start->next;
    }
}
/*****
** No more transistors to compare this time. Place the        **
** into the level1 device array.                              **
*****/

```

```

Pop();
}

```

```

/*****
** This function compares the transistor with all of the      **
** in the list to find a match. It calls itself recursively  **
** until all matches are found.                               **
*****/

```

```

Function combinep( startp,previst)
trans *startp, *previst;

```

```

{
    trans *compare;
    int    set;
    set=FALSE;
/*****
** Remove the transistor that now is part of a structure.      **
** If the transistor lists' head or tail pointer are to be    **
** deleted change the head or tail. Delete the used transistor **
** and decrease the transistor lists' length.                  **
*****/
compare = startp;
if(header_newp->length == 1) {
    set = TRUE;
    header_newp->length = 0;
    header_newp->head = NULL;
    startp->next=NULL;
}
else {
    if(startp== header_newp->head &&set!=TRUE) {
        header_newp->head = startp->next;
        header_newp->length--;
    }
    else {
        if(startp== header_newp->tail &&set!=TRUE) {
            header_newp->tail = previst;
            previst->next=NULL;
            header_newp->length--;
        }
        else {
            previst->next=previst->next->next;
            startp->next=NULL;
            header_newp->length--;
        }
    }
}

```

```

    }
}
}
/*****
** Go to the head of the list to begin the comparisons.      **
*****/
if(header_newp->head != NULL) {
    startp = header_newp->head;
    previst=startp;
}
else {
    startp = NULL;
}
while(startp != NULL)
{
/*****
** Are there any transistors that can be connected into a level1**
** device.                                                    **
*****/
    if (((strcmp(startp->source,compare->source))==0)&&
        ((strcmp(startp->source,"Vdd")!=0)) ||
        (((strcmp(startp->source,compare->drain))==0)&&
        ((strcmp(startp->source,"Vdd")!=0)) ||
        (((strcmp(startp->drain,compare->source))==0)&&
        ((strcmp(startp->source,"Vdd")!=0)) ||
        (((strcmp(startp->drain,compare->drain))==0)&&
        ((strcmp(startp->source,"Vdd")!=0)))
    {
/*****
** A match was found, place the transistor on the stack and    **
** find any transistors which connect to it.                  **
*****/
        Push(startp);
        combinep(startp,previst);
/*****
** The recursive call comes back to here. Reset the searching  **
** pointers to continue the search.                            **
*****/
        startp=header_newp->head;
    }
    else
    {
/*****
** No match was found increment the pointers.                **
*****/

```

```

    prevlst=starttp;
    starttp=starttp->next;
    }
}
/*****
** No more transistors to compare this time. Place the      **
** into the device array.                                   **
*****/
Pop();
}

/*****
** This function compares the transistor with all of the    **
** in the list to find a match. It calls itself recursively **
** until all matches are found.                             **
*****/

Function combinen(startn,previst)
trans *startn, *previst;

{
    trans *compare;
    int    set;
    compare = startn;
    set = FALSE;
/*****
** Does this n-type transistor connect to ground?          **
*****/
    if((strcmp("GND",startn->source))==0)
    {
        ground=TRUE;
    }
/*****
** Remove the transistor that now is part of a structure.   **
** If the transistor lists' head or tail pointer are to be **
** deleted change the head or tail. Delete the used transistor **
** and decrease the transistor lists' length.               **
*****/
if(header_newn->length == 1) {
    set = TRUE;
    header_newn->length = 0;
    header_newn->head = NULL;
    startn->next=NULL;

```

```

    }
else {
    if(startn== header_newn->head &&set!=TRUE) {
        header_newn->head = startn->next;
        header_newn->length--;
    }
else {
    if(startn== header_newn->tail &&set!=TRUE) {
        header_newn->tail = previst;
        previst->next=NULL;
        header_newn->length--;
    }
else {
        previst->next=previst->next->next;
        startn->next=NULL;
        header_newn->length--;
    }
}
}
}
/*****
** Go to the head of the list to begin the comparisons.      **
*****/
if(header_newn->head != NULL) {
    startn = header_newn->head;
    previst = startn;
}
else{
    startn = NULL;
}
while(startn != NULL)
{
/*****
** Are there any transistors that can be connected into a    **
** structure.                                                  **
*****/
    if(((strcmp(startn->source,compare->source))==0)&&
        ((strcmp(startn->source,"GND"))!=0))||
        (((strcmp(startn->source,compare->drain))==0)&&
        ((strcmp(startn->source,"GND"))!=0))||
        ((strcmp(startn->drain,compare->source))==0) ||
        ((strcmp(startn->drain,compare->drain))==0))
    {
/*****
** A match was found, place the transistor on the stack and   **
** find any transistors which connect to it.                  **
*****/

```

```

*****/
    Push(startn);
    combinen(startn,prev1st);
/*****
** The recursive call comes back to here. Reset the searching **
** pointers to continue the search. **
*****/
    startn=header_newn->head;
}

else
{
/* ****
** No match was found increment the pointers. **
*****/
    prev1st=startn;
    startn=startn->next;
}
}

/*****
** No more transistors to compare this time. Place the **
** into the level1 device array. **
*****/
Pop();
}

/*****
** This function compares the transistor with all of the **
** in the list to find a match. It calls itself recursively **
** until all matches are found. **
*****/

Function checkp(lo,hi)
int lo,hi;

{
    int pfound,nfound,i,j;
    int hi2nd,lasthi;
    trans *first, *prev1st, *second, *prev2nd;
    nfound=FALSE;
    first=header_newp->head;
    prev1st=first;
    second=header_newn->head;
    prev2nd=second;

```

```

while(first!=NULL) {
    pfound=FALSE;
    for(i=lo;i<=hi;i++) {
if((((strcmp(first->source,structure[numdevice][i]->source))==0)&&
    ((strcmp(first->source,"Vdd")!=0)))||
    (((strcmp(first->source,structure[numdevice][i]->drain))==0)&&
    ((strcmp(first->source,"Vdd")!=0)))||
    (((strcmp(first->drain,structure[numdevice][i]->source))==0)&&
    ((strcmp(first->source,"Vdd")!=0)))||
    (((strcmp(first->drain,structure[numdevice][i]->drain))==0)&&
    ((strcmp(first->source,"Vdd")!=0)))
    {
pfound=TRUE;
Push(first);
    combinep(first,prev1st);
hi2nd=numtrans-1;
for(j=hi;j<=hi2nd;j++) {
    while(second!=NULL) {
        if((((strcmp(second->source,structure[numdevice][i]->source))==0)||
            ((strcmp(second->source,structure[numdevice][i]->drain))==0)||
            ((strcmp(second->drain,structure[numdevice][i]->source))==0)||
            ((strcmp(second->drain,structure[numdevice][i]->drain))==0))
        {
stacknum=1;
Push(second);
        combinen(second,prev2nd);
nfound=TRUE;
second=header_newn->head;
prev2nd=second;
        lasthi=numtrans-1;
        }
    else {
        prev2nd=second;
second=second->next;
        }
        }
    }
}
if(pfound==TRUE) {
    first=header_newp->head;
    prev1st=first;
    break;
}
}

```

```

        if((pfound==FALSE)&&(first!=NULL)) {
            prev1st=first;
            first=first->next;
        }
    }
    if(nfound==TRUE) {
        checkp(hi2nd,lasthi);
    }
}

```

D. ISOMORPHISM VERIFICATION

```
/******  
* This function identifies if there exists isomorphism between the *  
* the transistors,doing this only numerically! *  
******/
```

Function isotrans()

```
{  
    if(p1 < (header_newp1->length)) { error(6); exit(0); }  
    if(p2 < header_newp2->length) { error(7); exit(0); }  
    if(n1 < header_newn1->length) { error(8); exit(0); }  
    if(n2 < header_newn2->length) { error(9); exit(0); }  
    error(42);  
}
```

```
/******  
* This function identifies if there exists isomorphism between the *  
* the inverters, numerically and qualitatively by checking assigned *  
* signatures like depth and width. *  
* (first->flag==0 indicates the first time for the device comparison ) *  
******/
```

Function isoinv()

```
{  
    int ok;  
    inv *first,*second;  
  
    ok = FALSE;  
    first=head1_inv->head;  
    second=head_inv->head;  
    if(head1_inv->length < head_inv->length) error(10);  
  
    while(second!=NULL) {  
        while(first!=NULL) {  
            if((first->flag==0)&&((strcmp(first->length,second->length))==0)&&  
                ((strcmp(first->width,second->width))==0)) {  
                first->flag=1; /* mark off the matched one */  
                first=head1_inv->head;  
                ok=TRUE;  
                break; /* found one now advance 2nd */  
            }  
        }  
        second=second->next;  
    }  
}
```

```

        else    first = first->next;
    } /* end while first*/
    if(ok!=TRUE) { error(11);  exit(0); }    /* no iso in inverter lists */
    else        second=second->next;
} /* end while second */
error(13);
}

```

```

/*****
* This function identifies if there exists isomorphism between the      *
* the 2 -phase clocked inverters, numerically and qualitatively        *
* by checking assigned signatures like depth and width.                  *
* (first->flag==0 indicates the first time for the device comparison ) *
*****/

```

Function iso2clinv()

```

{
    int ok;
    i2clinv *first,*second;

    ok = FALSE;
    first=head1_2clinv->head;
    second=head_2clinv->head;
    if(head1_2clinv->length < head_2clinv->length) error(14);

    while(second!=NULL) {
        while(first!=NULL) {
            if(((first->flag==0)&&((strcmp(first->length,second->length))==0)&&
                ((strcmp(first->width,second->width))==0)) {
                first->flag=1;
                first=head1_2clinv->head;
                ok=TRUE;
                break;
            }
            else    first = first->next;
        }
        if(ok!=TRUE) { error(15);  exit(0); }
        else        second=second->next;
    }
    error(17);
}

```

```

/*****
* This function identifies if there exists isomorphism between the
* the NOR2 gate structures, numerically and qualitatively, by
* checking assigned signatures like depth and width.
* (first->flag==0 indicates the first time for the device comparison )
*****/

```

Function isonor2()

```

{
    int ok;
    nor2 *first,*second;

    ok = FALSE;
    first=head1_nor2->head;
    second=head_nor2->head;
    if(head1_nor2->length < head_nor2->length) error(18);

    while(second!=NULL) {
        while(first!=NULL) {
            if(((first->flag==0)&&((strcmp(first->length,second->length))==0)&&
                ((strcmp(first->width,second->width))==0)) {
                first->flag=1;
                first=head1_nor2->head;
                ok=TRUE;
                break;
            }
            else first = first->next;
        }
        if(ok!=TRUE) { error(19); exit(0); }
        else second=second->next;
    }
    error(21);
}

```

```

/*****
* This function identifies if there exists isomorphism between the
* the NAND2 gate structures, numerically and qualitatively
* (first->flag==0 indicates the first time for the device comparison )
*****/

```

Function isonand2()

```

{

```

```

int ok;
nand2 *first,*second;

ok = FALSE;
first=head1_nand2->head;
second=head_nand2->head;
if(head1_nand2->length < head_nand2->length) error(22);

while(second!=NULL) {
    while(first!=NULL) {
        if((first->flag==0)&&((strcmp(first->length,second->length))==0)&&
            ((strcmp(first->width,second->width))==0)) {
            first->flag=1;
            first=head1_nand2->head;
            ok=TRUE;
            break;
        }
        else first = first->next;
    }
    if(ok!=TRUE) { error(23); exit(0); }
    else second=second->next;
}
error(25);
}

```

```

/*****
* This function identifies if there exists isomorphism between the
* the precharged NAND2 gates, numerically and qualitatively
* by checking assigned signatures like depth and width.
* (first->flag==0 indicates the first time for the device comparison )
*****/

```

Function isoprenand2()

```

{
    int ok;
    prenand2 *first,*second;

    ok = FALSE;
    first=head1_prenand2->head;
    second=head_prenand2->head;
    if(head1_prenand2->length < head_prenand2->length) error(26);

    while(second!=NULL) {
        while(first!=NULL) {

```

```

        if((first->flag==0)&&((strcmp(first->length,second->length))==0)&&
            ((strcmp(first->width,second->width))==0)) {
            first->flag=1;
            first=head1_prenand2->head;
            ok=TRUE;
            break;
        }
        else    first = first->next;
    }
    if(ok!=TRUE) { error(27); exit(0); }
    else    second=second->next;
}
error(29);
}

```

```

/*****
* This function identifies if there exists isomorphism between the
* the precharged NOR2 gates, numerically and qualitatively, by
* checking assigned signatures like depth and width.
* (first->flag==0 indicates the first time for the device comparison ) *
*****/

```

Function isoprenor2()

```

{
    int ok;
    prenor2 *first,*second;

    ok = FALSE;
    first=head1_prenor2->head;
    second=head_prenor2->head;
    if(head1_prenor2->length < head_prenor2->length) error(30);

    while(second!=NULL) {
        while(first!=NULL) {
            if((first->flag==0)&&((strcmp(first->length,second->length))==0)&&
                ((strcmp(first->width,second->width))==0)) {
                first->flag=1;
                first=head1_prenor2->head;
                ok=TRUE;
                break;
            }
            else    first = first->next;
        }
    }
}

```

```

        if(ok!=TRUE) { error(31); exit(0); }
        else      second=second->next;
    }
    error(33);
}

```

```

/*****
 * This function identifies if there exists isomorphism between the
 * the PASSGATE gate structures, numerically and qualitatively, by
 * checking assigned signatures like depth and width.
 * (first->flag==0 indicates the first time for the device comparison )
 *****/

```

```

Function isopass()
{
    int ok;
    pass *first,*second;

    ok = FALSE;
    first=head1_pass->head;
    second=head_pass->head;
    if(head1_pass->length < head_pass->length) error(34);

    while(second!=NULL) {
        while(first!=NULL) {
            if((first->flag==0)&&((strcmp(first->length,second->length))==0)&&
                ((strcmp(first->width,second->width))==0)) {
                first->flag=1;
                first=head1_pass->head;
                ok=TRUE;
                break;
            }
            else first = first->next;
        }
        if(ok!=TRUE) { error(35); exit(0); }
        else      second=second->next;
    }
    error(37);
}

```

```

/*****

```

```

* This function identifies if there exists isomorphism between the      *
* the precharged inverter gates, numerically and qualitatively, by      *
* checking assigned signatures like depth and width.                      *
* (first->flag==0 indicates the first time for the device comparison ) *
*****/

```

Function isopreinv()

```

{
    int ok;
    preinv *first,*second;

    ok = FALSE;
    first=head1_preinv->head;
    second=head_preinv->head;
    if(head1_preinv->length < head_preinv->length) error(38);

    while(second!=NULL) {
        while(first!=NULL) {
            if((first->flag==0)&&((strcmp(first->length,second->length))==0)&&
                ((strcmp(first->width,second->width))==0)) {
                first->flag=1;
                first=head1_preinv->head;
                ok=TRUE;
                break;
            }
            else first = first->next;
        }
        if(ok!=TRUE) { error(39); exit(0); }
        else second=second->next;
    }
    error(41);
}

```

```

/*****
* This function identifies if there exists isomorphism between the      *
* the abstract structures, numerically.                                  *
*****/

```

Function isostructures()

```

{
    trans *nodes;
    int ok,i,j,k;

```

```

ok=0;
for(i=1; i<=numdevice; i++)
{
    for(j=1; j<=numdevice1; j++) {
        if((pcount[i]==pcount1[j])&&(ncount[i]==ncount1[j])&&
            (p1count[i]==p1count1[j])&&(p2count[i]==p2count1[j])&&
            (n1count[i]==n1count1[j])&&(n2count[i]==n2count1[j])) ok=1;
    }

    if(ok!=1) {
        printf("NO-ISOMORPHISM! Better check the 'output'file\n");
        fprintf(fo,"Indication of NON-ISOMORPHISM ! The structure below,");
        fprintf(fo,"which is from the small file doesn't match exactly with");
        fprintf(fo,"any of the structures of the big file \n");
        for(k=1; k<=tcount[i]; k++) {
            nodes=structure[i][k];
            fprintf(fo,"%s %s %s %s \n",nodes->type,nodes->gate,
                nodes->source,nodes->drain);
        }
    }
}
}

```

E. OTHER FUNCTIONS

```
/*
 * This function examines the two files which are going to be checked for
 * isomorphism or just verification of any design if they are .sim files.
 * This is done by checking the first line of each file if it's according
 * to UCB format.
 */
```

```
#include "headers.h"
```

```
int proper(FP)
FILE *FP;
{
    int ok;
    char scan[5][15];
    ok=FALSE;
    strcpy(buffer,blank);
    fscanf(FP,"%s %s %s %s %s\n",scan[0],scan[1],scan[2],scan[3],scan[4]);
    printf("%s %s %s %s %s ",scan[0],scan[1],scan[2],scan[3],scan[4]);
    if ((strcmp (scan[0],"|")) != 0) error(0);
    if ((strcmp (scan[1],"units:")) != 0) error(1);
    if ((strcmp (scan[3],"tech:")) != 0) error(3);
    if ((strcmp (scan[4],"scmos")) != 0) error(4);
    ok = TRUE;
    if(FP == fp) printf("big file ok\n");
    if(FP == cp) printf("small file ok\n");
    return(ok);
}
```

```
/*
 * This function handles the error messages which are going to appear on
 * screen in case a typing error is made or a different format file is
 * used instead of UCB.
 */
```

```
error(index)
int index;
{
    fprintf(fo,"\n****%s****\n\n", msgtbl[index]);
    printf("****%s****\n",msgtbl[index]);
}
```

```

/*****
* This function creates header nodes for various lists.      *
*****/

```

```

Function create_head()
{
    header_newp=create();
    header_newn=create();
    header1_newp=create();
    header1_newn=create();
    header_newp1=create();
    header_newp2=create();
    header_newn1=create();
    header_newn2=create();
    header1_newp1=create();
    header1_newp2=create();
    header1_newn1=create();
    header1_newn2=create();
    head_inv=createinv();
    head1_inv=createinv();
    head_preinv=createpreinv();
    head1_preinv=createpreinv();
    head_pass=createpass();
    head1_pass=createpass();
    head_2clinv=create2clinv();
    head1_2clinv=create2clinv();
    head_nor2=createnor2();
    head1_nor2=createnor2();
    head_nand2=createnand2();
    head1_nand2=createnand2();
    head_prenand2=createprenand2();
    head1_prenand2=createprenand2();
    head_prenor2=createprenor2();
    head1_prenor2=createprenor2();
}

```

```

/*****
* This function creates the head and tail nodes for the list of transistors. *
*****/

```

```

Function head_type *create()

```

```

{
    head_type *temp;
    temp = MALLOC(head_type);
    temp->length = 0;
    temp->head = temp->tail = NULL;
    return(temp);
}

```

```

/*****
 * This function creates the head and tail nodes for the list of inverters.  *
 *****/

```

```

Function inve *createinv()
{
    inve *temp;
    temp = MALLOC(inve);
    temp->length = 0;
    temp->head = temp->tail = NULL;
    return(temp);
}

```

```

/*****
 * This function creates the head and tail nodes for the list of 2-input      *
 * precharged inverters.                                                         *
 *****/

```

```

Function preinve *createpreinv()
{
    preinve *temp;
    temp = MALLOC(preinve);
    temp->length = 0;
    temp->head = temp->tail = NULL;
    return(temp);
}

```

```

/*****
 * This function creates the head and tail nodes for the list of passgates.  *
 *****/

```

```

Function passg *createpass()

```

```

{
    passg *temp;
    temp = MALLOC(passg);
    temp->length = 0;
    temp->head=temp->tail=NULL;
    return(temp);
}

```

```

/*****
 * This function creates the head and tail nodes for the list of 2-phase
 * clock inverters.
 *****/

```

```

Function i2clinve *create2clinv()
{
    i2clinve *temp;
    temp= MALLOC(i2clinve);
    temp->length=0;
    temp->head=temp->tail=NULL;
    return(temp);
}

```

```

/*****
 * This function creates the head and tail nodes for the list of 2-input NOR
 * gates.
 *****/

```

```

Function nor2g *createnor2()
{
    nor2g *temp;
    temp= MALLOC(nor2g);
    temp->length=0;
    temp->head=temp->tail=NULL;
    return(temp);
}

```

```

/*****
 * This function creates the head and tail nodes for the list of 2-input NAND
 * gates.
 *****/

```

```

Function nand2g *createnand2()
{
    nand2g *temp;
    temp=MALLOC(nand2g);
    temp->length=0;
    temp->head=temp->tail=NULL;
    return(temp);
}

```

```

/*****
* This function creates the head and tail nodes for the list of precharged *
* 2-input NAND gates. *
*****/

```

```

Function prenand2g *createprenand2()
{
    prenand2g *temp;
    temp=MALLOC(prenand2g);
    temp->length=0;
    temp->head=temp->tail=NULL;
    return(temp);
}

```

```

/*****
* This function creates the head and tail nodes for the list of precharged *
* 2-input NOR gates. *
*****/

```

```

Function prenor2g *createprenor2()
{
    prenor2g *temp;
    temp=MALLOC(prenor2g);
    temp->length=0;
    temp->head=temp->tail=NULL;
    return(temp);
}

```

```

/*****
* This function creates the node for a structure of a transistor. *
*****/

```

```

Function trans *newnode()
{
    trans *newnodes;
    if(!(newnodes = MALLOC(trans))) {
        error(2);
        exit(1);
    }
    newnodes->next=NULL;
    newnodes->type=NULL;
    newnodes->gate=NULL;
    newnodes->source=NULL;
    newnodes->drain=NULL;
    newnodes->length=NULL;
    newnodes->width=NULL;
    return(newnodes);
}

```

```

/*****
* This function creates the node for a structure of an inverter.      *
*****/

```

```

Function inv *NewInvert()
{
    inv *newinvert;

    if(!(newinvert = MALLOC(inv))) {
        error(2);
        exit(1); }

    newinvert->flag=0;
    newinvert->input=NULL;
    newinvert->output=NULL;
    newinvert->length=NULL;
    newinvert->width=NULL;
    newinvert->next=NULL;
    return(newinvert);
}

```

```

/*****
* This function creates the node for a structure of a precharged      *
* inverter.                                                              *
*****/

```

```
Function preinv *NewPreinv()
```

```
{
    preinv *newpreinv;
    if(!(newpreinv = MALLOC(preinv))) {
        error(2);
        exit(1);
    }
    newpreinv->flag=0;
    newpreinv->input=NULL;
    newpreinv->output=NULL;
    newpreinv->phase=NULL;
    newpreinv->length=NULL;
    newpreinv->width=NULL;
    newpreinv->next=NULL;
    return(newpreinv);
}
```

```
/*
 * This function creates the node for a structure of a passgate.
 */
```

```
pass *NewPass()
```

```
{
    pass *newpass;
    if(!(newpass=MALLOC(pass))) {
        error(2);
        exit(1);
    }
    newpass->flag=0;
    newpass->terminal1=NULL;
    newpass->terminal2=NULL;
    newpass->php=NULL;
    newpass->phn=NULL;
    newpass->length=NULL;
    newpass->width=NULL;
    newpass->next=NULL;
    return(newpass);
}
```

```
/*
 * This function creates the node for a structure of 2-phase clocked
 * inverter.
 */
```

```

*****/

i2clinv *New2clinv()
{
    i2clinv *new2clinv;
    if(!(new2clinv=MALLOC(i2clinv))) {
        error(2);
        exit(1);
    }
    new2clinv->flag=0;
    new2clinv->input=NULL;
    new2clinv->output=NULL;
    new2clinv->php=NULL;
    new2clinv->phn=NULL;
    new2clinv->length=NULL;
    new2clinv->width=NULL;
    new2clinv->next=NULL;
    return(new2clinv);
}

/*****
* This function creates the node for a stucture of 2-input NOR gate. *
*****/

nor2 *Newnor2()
{
    nor2 *newnor2;
    if(!(newnor2=MALLOC(nor2))) {
        error(2);
        exit(1);
    }
    newnor2->flag=0;
    newnor2->input1=NULL;
    newnor2->input2=NULL;
    newnor2->output=NULL;
    newnor2->length=NULL;
    newnor2->width=NULL;
    newnor2->next=NULL;
    return(newnor2);
}

/*****
* This function creates the node for a structure of a 2-input NAND gate.*
*****/

```

```

*****/

nand2 *Newnand2()
{
    nand2 *newnand2;
    if(!(newnand2=MALLOC(nand2))) {
        error(2);
        exit(1);
    }
    newnand2->flag=0;
    newnand2->input1=NULL;
    newnand2->input2=NULL;
    newnand2->output=NULL;
    newnand2->length=NULL;
    newnand2->width=NULL;
    newnand2->next=NULL;
    return(newnand2);
}

/*****
 * This function creates the node for a structure of a precharged NAND  *
 * gate.                                                                *
 *****/

prenand2 *Newprenand2()
{
    prenand2 *newprenand2;
    if(!(newprenand2=MALLOC(prenand2))) {
        error(2);
        exit(1);
    }
    newprenand2->flag=0;
    newprenand2->input1=NULL;
    newprenand2->input2=NULL;
    newprenand2->output=NULL;
    newprenand2->phase=NULL;
    newprenand2->length=NULL;
    newprenand2->width=NULL;
    newprenand2->next=NULL;
    return(newprenand2);
}

/*****

```

```

* This function creates the node for a structure of a precharged NOR *
* gate. *
*****/

```

```

prenor2 *Newprenor2()
{
    prenor2 *newprenor2;
    if(!(newprenor2=MALLOC(prenor2))) {
        error(2);
        exit(1);
    }
    newprenor2->flag=0;
    newprenor2->input1=NULL;
    newprenor2->input2=NULL;
    newprenor2->output=NULL;
    newprenor2->phase=NULL;
    newprenor2->length=NULL;
    newprenor2->width=NULL;
    newprenor2->next=NULL;
    return(newprenor2);
}

```

```

/*****
* This function prints all transistors according to their properties *
* used in the entire program. That means there are four sets of *
* transistors which are printed : *
* 1. P transistors without Vdd in the source field. *
* 2. P transistors with Vdd in the source field. *
* 3. N transistors without GND in the source field. *
* 4. N transistors with GND in the source field. *
*****/

```

```

Function printtrans()
{
    int i,j,k,l;
    trans *node1,*node2,*node3,*node4;
    node1 = header_newp1->head;
    node2 = header_newp2->head;
    node3 = header_newn1->head;
    node4 = header_newn2->head;
    fprintf(fo," \n");
    if(header_newp1->length!=0){
        fprintf(fo,"The p-type transistors not connected to Vdd in this file are:\n");
    }
}

```

```

    }
    for(i=1; i<= header_newp1->length; i++) {
        fprintf(fo,"P1 No%d - GATE:%s SOURCE:%s DRAIN:%s \n",i,node1->gate,
            node1->source,node1->drain);
        node1=node1->next;
    }
    fprintf(fo," \n");
    if(header_newp2->length!=0){
        fprintf(fo,"The p-type transistors connected to Vdd in this file are:\n");
    }
    for(i=1; i<= header_newp2->length; i++) {
        fprintf(fo,"P2 No%d - GATE:%s SOURCE:%s DRAIN:%s \n",i,node2->gate,
            node2->source,node2->drain);
        node2=node2->next;
    }
    fprintf(fo," \n");
    if(header_newn1->length!=0){
        fprintf(fo,"The n-type transistors not connected to GND in this file are:\n");
    }
    for(i=1; i<=header_newn1->length; i++) {
        fprintf(fo,"N1 No%d - GATE:%s SOURCE:%s DRAIN:%s \n",i,node3->gate,
            node3->source,node3->drain);
        node3=node3->next;
    }
    fprintf(fo," \n");
    if(header_newn2->length!=0){
        fprintf(fo,"The n-type transistors connected to GND in this file are:\n");
    }
    for(i=1; i<=header_newn2->length; i++) {
        fprintf(fo,"N2 No%d - GATE:%s SOURCE:%s DRAIN:%s \n",i,node4->gate,
            node4->source,node4->drain);
        node4=node4->next;
    }
}
}

```

```

/*****
* This function prints all inverters which have been identified.  *
*****/

```

```

print_inv()
{
    int i;
    inv *node_inv;
    node_inv = head_inv->head;

```

```

fprintf(fo," \n");
if(head_inv->length!=0){
fprintf(fo,"The INVERTERS that exist in this file are:\n");
}
for(i=1; i<=head_inv->length; i++) {
    fprintf(fo,"INVERTER No%d - INPUT: %s OUTPUT: %s LENGTH: %s WIDTH: %s \n",
        i,node_inv->input,node_inv->output,node_inv->length,node_inv->width);
    node_inv = node_inv->next;
}
}

```

```

/*****
* This function prints all precharged inverters which have been identified. *
*****/

```

```

print_preinv()
{
    int i;
    preinv *node_preinv;
    node_preinv=head_preinv->head;
    fprintf(fo," \n ");
    if(head_preinv->length!=0){
        fprintf(fo,"The PRECHARGED INVERTERS that exist in this file are:\n");
    }
    for(i=1; i<=head_preinv->length; i++) {
        fprintf(fo,"PRECH.INVERTER No%d - INPUT:%s OUTPUT:%s PHASE:%s \n",
            i,node_preinv->input,node_preinv->output,node_preinv->phase);
        fprintf(fo,"                LENGTH: %s WIDTH: %s \n",
            node_preinv->length,node_preinv->width);
        node_preinv = node_preinv->next;
    }
}

```

```

/*****
* This function prints all passgates which have been identified. *
*****/

```

```

print_pass()
{
    int i;
    pass *node_pass;
    node_pass=head_pass->head;
    fprintf(fo," \n");
}

```

```

    if(head_preinv->length!=0){
    fprintf(fo,"The PASSGATES that exist in this file are:\n");
    }
    for(i=1; i<=head_pass->length; i++) {
        fprintf(fo,"PASSGATE No %d - TERM1: %s TERM2: %s PHP: %s PHN: %s \n",
            i,node_pass->terminal1,node_pass->terminal2,node_pass->php,node_pass->phn);
        fprintf(fo,"                LENGTH: %s WIDTH: %s \n", node_pass->length,
            node_pass->width);
        node_pass=node_pass->next;
    }
}

```

```

/*****
* This function prints all 2-phase clocked inverters which have been
* identified.
*****/

```

```

print_2clinv()
{
    int i;
    i2clinv *node_2clinv;
    node_2clinv=head_2clinv->head;
    fprintf(fo," \n");
    if(head_2clinv->length!=0){
    fprintf(fo,"The 2-PHASE CLOCK INVERTERS that exist in this file are:\n");
    }
    for(i=1; i<=head_2clinv->length; i++) {
        fprintf(fo,"2 PH. CLOCK INV. No%d - INPUT:%s OUTPUT:%s PHP:%s PHN:%s \n",i,
            node_2clinv->input,node_2clinv->output,node_2clinv->php,node_2clinv->phn);
        fprintf(fo,"                LENGTH:%s WIDTH:%s \n",
            node_2clinv->length,node_2clinv->width);
        node_2clinv=node_2clinv->next;
    }
}

```

```

/*****
* This function prints all 2-input NOR gates which have been identified.
*****/

```

```

print_nor2()
{
    int i;

```

```

nor2 *node_nor2;
node_nor2=head_nor2->head;
fprintf(fo," \n");
if(head_nor2->length!=0){
fprintf(fo,"The 2-input NOR gates that exist in this file are:\n");
}
for(i=1; i<=head_nor2->length; i++) {
    fprintf(fo,"NOR-2 No%d - INPUT1:%s INPUT2:%s OUTPUT:%s \n",
    i,node_nor2->input1,node_nor2->input2,node_nor2->output);
    fprintf(fo,"                LENGTH:%s WIDTH:%s \n",
    node_nor2->length,node_nor2->width);
    node_nor2=node_nor2->next;
}
}

```

```

/*****
* This function prints all 2-input NAND gates which have already been *
* identified. *
*****/

```

```

print_nand2()
{
    int i;
    nand2 *node_nand2;
    node_nand2=head_nand2->head;
    fprintf(fo," \n");
    if(head_nand2->length!=0){
    fprintf(fo,"The 2-input NAND gates that exist in this file are:\n");
    }
    for(i=1; i<=head_nand2->length; i++) {
        fprintf(fo,"NAND-2 No%d - INPUT1:%s INPUT2:%s OUTPUT:%s \n",
        i,node_nand2->input1,node_nand2->input2,node_nand2->output);
        fprintf(fo,"                LENGTH:%s WIDTH:%s \n",
        node_nand2->length,node_nand2->width);
        node_nand2=node_nand2->next;
    }
}

```

```

/*****
* This function prints all precharged 2-input NAND gates which have already *
* been identified. *
*****/

```

```

print_prenand2()
{
    int i;
    prenand2 *node_prenand2;
    node_prenand2=head_prenand2->head;
    fprintf(fo," \n");
    if(head_prenand2->length!=0){
        fprintf(fo,"The 2-input PRECHARGED NAND gates that exist in this file are:\n");
    }
    for(i=1; i<=head_prenand2->length; i++) {
        fprintf(fo,"PRECH. NAND-2 No%d - INPUT1:%s INPUT2:%s OUTPUT:%s \n",
            i,node_prenand2->input1,node_prenand2->input2 node_prenand2->output);
        fprintf(fo,"
                                PHASE:%s LENGTH:%s WIDTH:%s \n",
            node_prenand2->phase,node_prenand2->length node_prenand2->width);
        node_prenand2=node_prenand2->next;
    }
}

```

```

/*****
* This function prints all precharged 2-input NOR gates which have already *
* been identified. *
*****/

```

```

print_prenor2()
{
    int i;
    prenor2 *node_prenor2;
    node_prenor2=head_prenor2->head;
    fprintf(fo," \n");
    if(head_prenor2->length!=0){
        fprintf(fo,"The 2-input PRECHARGED NOR gates that exist in this file are:\n");
    }
    for(i=1; i<=head_prenor2->length; i++) {
        fprintf(fo,"PRECH. NOR-2 No%d - INPUT1:%s INPUT2:%s OUTPUT:%s \n",
            i,node_prenor2->input1,node_prenor2->input2 node_prenor2->output);
        fprintf(fo,"
                                PHASE:%s LENGTH:%s WIDTH:%s \n",
            node_prenor2->phase,node_prenor2->length,node_prenor2->width);
        node_prenor2=node_prenor2->next;
    }
}

```

```

/*****
* This function prints all the transistors which were not grouped in *
* certain gate structures and they considered as abstract structures.*
*****/

print_structures()
{
    int i,j;
    trans *node;
    if(numdevice!=0){
        fprintf(fo,"\n*****Other STRUCTURES *****\n");
        for(i=1; i<= numdevice; i++) {
            fprintf(fo,"\nThere are %d transistors in this device.\n",tcount[i]);
            for(j=1; j<= tcount[i]; j++) {
                node = structure[i][j];
                fprintf(fo,"%s %s %s %s\n",node->type,node->gate,node->source,node->drain);
            }
        }
    }
}

```

```

/*****
* This function prints all the transistors which were not grouped in *
* certain gate structures and they considered as abstract structures.*
*****/

print_structures1()
{
    int i,j;
    trans *node;

    if (numdevice1!=0){
        fprintf(fo,"\n*****Other STRUCTURES *****\n");
        for(i=1; i<= numdevice1; i++) {
            fprintf(fo,"\nThere are %d transistors in this device.\n",tcount1[i]);
            for(j=1; j<= tcount1[i]; j++) {
                node = structure1[i][j];
                fprintf(fo,"%s %s %s %s\n",node->type,node->gate,node->source,node->drain);
            }
        }
    }
}

```

```

/*****
* This function provides the error message for the structure
* procedure.
*****/

print_error()
{
    printf("There is a transistor connected to Vdd with no path to GND.\n");
    printf(" Please verify your circuit.\n");
    fprintf(fo,"There is a transistor connected to Vdd with no path to GND.\n");
    fprintf(fo," Please verify your circuit.\n");
}

```

REFERENCES

1. Jacobs, H., "Verification of a Second Generation 32-bit Microprocessor," *Computer*, April 1986, pp. 64-70.
2. Ablasser, I., and Jager, U., "Circuit Recognition and Verification Based on Layout Information," *Proceedings of the 18th ACM/IEEE Design Automation Conference*, pp. 684-689, June 1981.
3. Tucker, Thomas W., and Gross, Jonathan L., *Topological Graph Theory*, p. 1-2, John Wiley & Sons, Inc., 1987.
4. Trudeau, Richard J., *Dots and Lines*, pp.21-42, Kent State University Press, 1976.
5. Tutte, W. T., *Connectivity in Graphs*, pp. 46-47, Oxford University Press, 1966.
6. Garey, M., and Johnson, D., *Computer and intractability: A guide to the theory of NP-completeness*, Freeman and Co., San Fransisco, 1978.
7. Beineke, Lowell W., and Wilson, Robin J., *Selected Topics in Graph Theory*, pp. 422, Academic Press, Inc., 1978.
8. Read, R. C., and Corneil, D. G., "The Graph Isomorphism Dicease," *Journal of Graph Theory*, v. 1, pp. 339-363, 1977.
9. University of California Berkeley Report No UDB/CSD 86/272, *1986 VLSI Tools: Still More Works by the Original Artists; "Berkeley CAD Tools User's Manual"* by Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, December 1986.
10. Swisher, Joel V., *Circuit Recognition of VLSI Layouts*, Master's Thesis, Naval Post-graduate School, Monterey, California, December 1989.

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Superintendent, Naval Postgraduate School Attn: Professor Chyan Yang, Code EC/Ya Naval Postgraduate School Monterey, California 93943-5000	6
5.	Superintendent, Naval Postgraduate School Attn: Professor Mitchell L. Cotton, Code EC/Co Naval Postgraduate School Monterey, California 93943-5000	1
6.	Hellenic Navy General Staff 2nd Branch, Education Department Stratopedon Papagou, Holargos Athens 15561, GREECE	4
7.	LT Emmanouil N. Zagourakis, H.N. 184 Patission St. Athens 11257, GREECE	2